Computational Verification of Security Requirements

Bibu, Gideon Dadik

Award date:
2014

Awarding institution:
University of Bath

Computational Verification of Security Requirements

submitted by

Gideon Dadik Bibu

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

September 2013

Abstract

One of the reasons for persistence of information security challenges in organisations is that security is usually seen as a technical problem. Hence the emphasis on technical solutions in practice. However, security challenges can also arise from people and processes. We therefore approach the problem of security in organisations from a socio-technical perspective and reason that the design of security requirements for organisations has to include procedures that would allow for the design time analysis of the system behaviour with respect to security requirements.

In this thesis we present a computational approach to the verification and validation of elicited security requirements. This complements the existing approaches of security requirements elicitation by providing a computational means for reasoning about security requirements at design time. Our methodology is centred on a deontic logic inspired institutional framework which provides a mechanism to monitor the *permissions*, *empowerment*, and *obligations* of actors and generates violations when a security breach occurs.

We demonstrate the functionality of our approach by modelling a practical scenario from health care domain to explore how the institutional framework can be used to develop a model of a system of interacting actors using the action language InstAL. Through the application of the semantics of answer set programming (ASP), we demonstrate a way of carrying out verification of security requirements such that it is possible to predict the effect of certain actions and the causes of certain system states. To show that our approach works for a number of security requirements, we also use other scenarios to demonstrate the analysis of confidentiality and integrity requirements.

From human factor point of view compliance determines the effectiveness of security requirements. We demonstrate that our approach can be used for management of security requirements compliance. By verifying compliance and predicting non-compliance and its consequences at design time, requirements can be redesigned in such a way that better compliance can be achieved.

Acknowledgements

It has been a long journey so far. I have heard it said several times that PhD is a long marathon which requires determination to stay motivated in addition to stamina and focus. Indeed it takes all these to weather through a PhD studies. With this said it is obvious that this feat could not have been achieved alone.

I give thanks and praise to my Father in heaven, the Almighty God for making this possible all the way from the beginning to the end. It was almost an impossibility due to the funding problem but He came through at the right time. To Him alone be all the praise and glory.

Special thanks to my wife Rahab for her support, encouragement and prayers. Also to our lovely and wonderful kids Retnyit, Zughumnan and Kwoopnan. Thank you for those prayers and encouragements that you always give. You are really blessings to us. Thank you for enduring all those evenings and week-ends when daddy has to be at work at the expense of your play time. God bless you and keep you.

To my great, wonderful and amiable supervisors, Dr Julian Padget, Prof James Davenport and Dr Marina De Vos, I want to say a big thank you for the direction and pushing me to my limits in several ways. I am indeed better off in many ways than when I first met you. I salute Julian's patience as I tried to find my way round the concepts that have brought me to this point today. Thank you to James for his insight and constructive criticisms on security issues at the early stage of my work. Marina who had been unofficially provision some level of supervision even before she officially became my second-second supervisor, thank you for those once in a while thought baits you throw at me, especially with ASP and InstAL. I also want to thank Dr Tina Balke for helping out with InstAL

issues.

Life in the PhD lab can be a rather boring time. Thanks to those who spice it up in many different ways. Particularly to Mesar Hameed thank you for those walk-out times and support with technical issues. To Saeid Pourroostaei Ardakani, Pawitra Chiravirakul, Zohreh Shams, Dr Fabio Nemetz and Dr Joao Duro my seat row mates thank you for your support.

My research experience cannot be complete without the time I spent in NII, Japan. I want to acknowledge Professor Nobukazu Yoshioka for his great influence on my research. The time I spent working with him at NII helped me sharpen my research focus. I acknowledge NII for the opportunity.

Thanks to my parents, siblings and in-laws for your prayers and encouragement. I thank friends in JKC Bath and the Olukus for their support and great measure of love. The Lord shall uphold you all in Jesus name.

I specially acknowledge the Petroleum Technology Development Fund (PTDF) Nigeria for sponsoring my study fully for three years. Without your funding, I would still have been dreaming about doing a PhD outside Nigeria. I also acknowledge other funding bodies including CISN hardship fund for the financial support granted to me towards the end of my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The mantra of any good security engineer is: 'Security is a not a product, but a process.' It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together.

Bruce Schneier

The unrivalled developments in the computer systems in recent times have seen both individuals and organisations become more dependent on computer based technologies, especially the information system infrastructure, for their daily uses. Organisations particularly now depend on information technology for their business operations and processes. As a result, such organisations are faced with the challenge of security vulnerabilities as more and more highly sensitive content is processed and managed electronically.

Security can be viewed from different perspectives and as such definition of security is usually derived from the perspective in which it is viewed. Security has meant different things to different people ranging from physical security of servers and workstations from those who might try to steal them, security of data from

viruses and worms and the means by which they may be kept from entering a network, security of data from hackers and miscreants, to providing even comfort that comes in knowing that you can restore files if a user accidentally deletes them could also be seen as security (Pastore and Dulaney, 2006). However, regardless of the context in which security is defined, the goals of any information system security are;

- **Prevention:** This is the traditional core of computer or information security. It is aimed at stopping security breaches from occurring.
- **Detection:** refers to identifying events *when* they occur. This can be difficult in some situations since attack on a system may occur over a long period before it is successful. Incident detection involves identifying the assets under attack, how the incidence occurred, and who carried (or is still carrying) it out. Detection should be an ongoing going activity in an organisation.
- **Response:** refers to developing strategies and techniques to deal with an attack or loss. This involves having some well thought-out and tested plans to be used to respond, restore operations, and neutralise threats. It is better to have these in hand than trying to create them when the attack has already occurred.

These goals create the framework for developing and maintaining security for information systems. Security is generally seen as the ability of a system to protect information and system resources with respect to $confidentiality$, $integrity$, and $availability$ (CIA)(Olivier, 2002). Some researchers and security professionals have also made a case for the inclusion of $accountability$ as a security goal that needs to be met but not covered by the CIA trio (Haley et al., 2006; Pastore and Dulaney, 2006; Anderson, 2008). Each of these security sub goals presents different challenges which define the security design goals of any organisation's information system. Also, it is worth noting that there is usually tension between these security goals. For instance, between confidentiality and availability which we are not going into in this thesis but would like to note that such a tension makes the design of security requirements challenging.

Security of information has remained a challenge in practice. Despite the adoption and implementation of various security technologies, organisations still remain vulnerable to security incidents (Fry and Nystrom, 2009; DofBIS-UK, 2013). According to the 2013 information security breach survey report commissioned by the UK department of business innovation and skills (DofBIS-UK, 2013, pp. 2–3), organisations' spending on security has increased from the previous years due to the increase in awareness of security challenges and the need to invest more on security. However, contrary to expectations, the number of security breaches affecting UK businesses continues to increase rather than reduce. This rise in security breach is most notable for small organisations who are now experiencing incident levels previously associated only with larger organisations with affected organisations experiencing roughly 50% more breaches on average than the previous year. These breaches are not without associated financial implications to the organisations. The report confirms the rise in the cost of individual breaches surveyed with several breaches costing more than £1m in the year under survey. In total, this cost to the UK organisations is in the order of billions of pounds per annum which is roughly tripled over the previous year.

Sources of the reported security breaches span from external attacks to internal attacks, underlining the fact that attacks could be launched from both inside and outside the organisation. External threats are usually addressed by creating a "perimeter" around the organisation's assets such as firewalls, which provide defences against the perceived external attacks. This does not solve the insider threat problem which is more subtle than the external threat problem, in the sense that insider threat manifests itself in many ways, including through user behaviours that violate security policies. These behaviours could either be malicious or non-malicious, however, whatever the intention, these behaviours always have negative impact on the organisation. A malicious insider is potentially more dangerous than an outside attacker. This is because an insider has legitimate and privileged access to information resources, practical knowledge of the organisation and its processes, and knowledge of the location of valuable and critical assets. An important step in mitigating the risks posed by insider attacks is to carefully construct an enterprise-wide security policy that addresses usage and security issues (Mike and Kemp, 2005; Colwill, 2009) in addition to the implementation

of necessary security threat mitigation mechanisms.

Looking at the internal attacks which is the focus of this thesis, serious security breaches were reported to have been due to the multiple internal sources including failures in technology, processes, and people with staff playing key role in many breaches. The report shows that 36% of the worst security breaches were caused by unintentional human error with a further 10% by deliberate misuse of systems by staff. Particularly in small organisations surveyed, 57% of them suffered staff related security breaches which is 45% rise from a year ago. This report is also supported by the Ponemon report (Institute, 2012) which emphasised that organizations *"must address how employees factor into overall data security"*. All these go to point out the fact that organisations are struggling to keep up with security threats despite increased attention given to security in IT budget compared with previous years.

Remedies for security vulnerabilities and breaches tend to focus on technical mechanisms, including firewalls and implementation of encryption, which are often designed and implemented with little consideration for the needs and characteristics of the end users, including network administrators and managers. However, organisations are made up of human individuals who interact with each other and with various organisational resources such as information and data as they carry out their duties. As such, they have the tendency to exhibit behaviours that circumvent the security efforts of such organisations. For instance, requiring a user to use a complex password may result in the user writing a note with the password attached to the computer screen.

The potential weakness of technical solutions and the danger of focusing solely on technical solutions have been highlighted by security experts. Schneier (Schneier, 2000) for instance expressed this as follows:

> *Computer security is difficult (maybe even impossible), but imagine for a moment that we've achieved it. Unfortunately, this still isn't enough. For this miraculous computer system to do anything useful, it is going to have to interact with users in some way, at some time, for some reason. And this interaction is the biggest security risk of them all. People often represent the weakest link in the security chain*

and are chronically responsible for the failure of security systems.

A similar message is provided by Mitnick and Simon (Mitnick and Simon, 2002):

> *A company may have purchased the best security technologies that money can buy, trained their people so well that they lock up all their secrets before going home at night, and hired building guards from the best security firm in the business. The company is still totally vulnerable... the human factor is truly security's weakest link.*

These two quotes attribute the weakest security link to the human factor. The lack of consideration for human factors may create situations where people have to circumvent the security mechanisms and procedures in order to perform their job efficiently. Also, if security processes "get in the way" of human activities or impose tedious overheads, they are more likely to be circumvented. For instance, a requirement for the user to perform a security scan on the system at login may be circumvented by the user who needs to get some work done urgently. Likewise security policies that are formed without sufficient consideration for implementation issues may be difficult to implement by the users.

Therefore, technical controls in isolation are not enough to protect organisations since with the appropriate motivation and time, human beings will find their way around most technical controls (Colwill, 2009; Kraemer et al., 2009). It is important to design security systems and processes that consider the structure of the organization and the interactions of the various data users and handlers within the organization, i.e. to adopt an approach based on the discipline of organizational behaviour and human factor engineering. Such an approach can help in understanding behaviours of end users and the factors that affect their behaviours which will in turn influence the definition of appropriate security requirements.

Another reason for the prevalence of security breaches is that security requirements are frequently not considered at system design time but rather added to the system later in the software life-cycle. Consideration of security in the system development life cycle is considered essential to implementing and integrating a comprehensive strategy for managing security risks for all information technology

assets in an organisation (Kissel et al., 2008). Therefore, identifying and analysing security requirements should be an important element of the software engineering process. This would lead to the early identification of security loopholes in the system thereby helping in the provision of appropriate mitigations.

We therefore see a potential here in using the approach of monitoring events in organisations to address security problems that are not due to the insecurity of the underlying system infrastructure, but security threats that are due to the vulnerabilities that could not be elicited at system design time. These behaviour related threats include information leakage, system failures, abuse of privileges, and violation of security policies. Our solution approach will involve taking into account the social structure that exists in an organisation, the security policies and the interactions between actors. We illustrate our approach using suitable scenarios that capture the traditional security goals in the sections that follow.

Before we go further, we need to define what we mean by *system* in this thesis.

Definition 1.

We take the term System to include the software, hardware, and the people who interact with these software and hardware by way of usage.

This definition is consistent with common usage within the requirements engineering community, of which security requirements are part. For example, *"...we use 'system' only to refer to a general artefact that might have both manual and automatic components, such as an 'airline reservation system'."* - Zave and Jackson (1997). Also, van Lamsweerde (2000) describes a target *system* as not just a piece of software, but also comprising the *environment* that surrounds it. Therefore as requirements engineering is aimed at providing detailed and relevant information about the requirements that a system must satisfy, security requirements engineering seeks to establish the adequate requirements that if respected, would lead to satisfaction of system's security goals.

We also use the term *policy* to refer to *security policy*. Policy has been used in many ways to address security issues consisting of confidentiality, integrity, and availability. Policies can be sets of rules that define choices in behaviour

of participants in a system in terms of the conditions under which predefined operations or actions can be invoked. Security policies provide the first step in preventing insider abuse in organisations as expected security behaviours are usually presented in the form of some high level security policies. However, the problem with policies is that compliance cannot be guaranteed and hence the likelihood for the security threats to persist, despite the existence of policies that should have ensured proper behaviour by users. With this in mind, it is important that security policy designers are able to verify that the policy is consistent with the operations of the organisation so that everyday organisational processes do not stand in the way of compliance. It is also necessary that security designers are able to verify the effect of non-compliance to security policies on the state of the organisation. This would help in refining the policy before it is deployed.

Many approaches have been proposed for the elicitation of security requirements at system design time (e.g. Sindre and Opdahl (2005); Pauli and Xu (2005); McDermott and Fox (1999); Liu et al. (2003); Mouratidis (2011); van Lamsweerde (2004)). However, one missing thing is that they do not provide for a way of formal reasoning about the requirements, as a result they are not able to say anything about the effect of the constraints elicited on the system functionality. The importance of eliciting the appropriate security requirements is non trivial. Research in elicitation of security requirements has gone from the use of traditional requirements engineering approaches to multi-agent system (MAS) and socio-technical systems (STS) approaches that aim at recognising the interaction of heterogeneous actors in elicitation of security requirements. This thesis is more aligned towards the MAS and STS approaches and we aim at enhancing the process of elicitation of system security requirements through the provision of a mechanism that allows for reasoning about the requirements elicited at design time, taking note particularly, of the fact that the actors in the system could behave in many different ways that could impact on the security of the system at run time.

1.1 Security, Logic and Social Norms

Today, the implicit philosophy of computer and information security is largely based on the notion of containment which is taken from the analogy of the kind of

protection offered for physical things such as fences, gates, locks etc (Gasser, 1988; Pieters, 2011). This implies that the assets to be protected need to be separated from the environment. This separation is provided for by the use of security boundaries such as firewalls which filter all inbound and outbound traffic on an organisation's network, hence providing protection from potentially harmful transmissions. In this fortress-based approach to security, the fortress may be robust against external threats but weak against those emanating from within the security perimeter. The latter, though very potent threats in today's world of information security, has been largely left unexamined as much information security endeavours have been devoted to the former.

In his work, Pieters (2011) further points out how unsatisfactory this implicit philosophy is in the current age of increased connectivity. Therefore rather than having security of information systems thought of as rational design choices only, the alternative idea is to think of information security as a co-evolution of social and technical mechanisms, hence the socio-technical perspective adopted in this thesis. Central to this perspective is the idea that; i) information security is not merely a design problem, as external forces also participate in shaping the threats to and protection of information system ii) humans play a central role in the security of information systems, both as attackers and defenders. Therefore the notion of fortress-based security can no longer suffice because

i) there is an increasing demand to access organisation's assets from outside the organisation's physical parameter as organisations undergo transformations in order to keep up with today's increasingly competitive environment. This demand for external access is warranted by outsourcing of services and computing resuources as occassioned by technologies enabled by cloud computing (Armbrust et al., 2010; Mell and Grance, 2011; Colwill, 2009)

ii) there is the problem of insider threat. With respect to information access, the *insider* is considered to be an individual who is currently or at one time have been authorized to access an organization's information system, data, or network. Such authorization implies a degree of trust in the individual as posited in the RAND report (Anderson and Brackney, 2004, p. xi) and Bishop (Bishop, 2005; Bishop and Gates, 2008). The *insider threat* therefore

refers to any act that trusted insiders might carry out in violation of security policies such that it causes harm to the organization or benefits the individual (Greitzer et al., 2008; Probst et al., 2007).

These have challenged the notion of fortress-based security, and solutions need to be developed that include human behaviours.

The philosophy of security design is often an adversarial one in which one treats the other as a potential enemy and this goes against our social and ethical predispositions. Organisational culture has social and ethical predispositions encoded into it. Also a family of culture-related drivers such as the organisational ideology, beliefs, rituals and myths; motivate and shape the organisational practices (Pettigrew, 1979). Attackers' awareness of these predispositions enables them to exploit the resultant weaknesses by means of social engineering. For example, since people do not regard Santa Claus as a malevolent symbol, they are not likely to immediately associate it with malicious behaviour, hence, dressing up as Santa Claus could work well for accessing restricted areas. From the perspective of security policy compliance for instance, situations usually arise in which actors face contradictory expectations on their behaviour, as expressed in organisational or social norms.

Research in psychology has shown that norms motivate human action and people tend to do what is socially approved as well as what is popular. Putting this in perspective, a distinction has been made between two kinds of norms at work place which could affect the security of the organisation in different ways: *injunctive norms* which involves perceptions of which behaviours are typically approved or disapproved, and *descriptive norms* which has to do with perception of which behaviours are typically performed (Cialdini, 2003). Thus, people's actions may be based on what they perceive to be an approved behaviour or what they perceive to be a common behaviour in which case, there is no expectation of approval. In security, both types of norms may improve or worsen security. For instance, the perception that quick delivery of results will be approved, even if security procedures are evaded can impact on security negatively. This happens when people think that to get their job done quickly is important and perceive that it will be an approved behaviour.

An example of how norms impact on security in organisations is the data loss incident at the UK's HM Revenue and Customs (HMRC) as analysed in the Poynter report (Poynter, 2008). In this incident, two compact disks (CDs) containing personal details of 25 million people were lost in transit between HM Revenue and Customs (HMRC) and the National Audit office (NAO). The disks contained personal records of families claiming child benefit, being financial aid given to families with children in the UK. The data which was comprised also of address and bank details of families under this system was being transferred as part of the external audit of the cases managed under this benefit system. Because child benefit is widely used by UK society, the event caused a serious public reaction. This helped in bringing into sharp focus the issues related to how citizens' personal data are managed by organisations. The Poynter report was commissioned in response to the data loss. The report detailed the events of the incident which led to the loss of two CDs. In addition, the root cause analysis of the issues that led to the incident was also presented in the report. A number of issues revealed in this report related to security policy which include:

i) the difficulty in interpretation of sufficient authorisation to release the CDs,
ii) the inadequate definition of obligations under the policy,
iii) poor specification of security controls for data in transit, and
iv) the unenforceable nature of the policy regarding the method of CD transport.

In this report, two significant issues relating to culture were also identified:

i) staff below a certain grade prioritise operational requirements over security requirements, and

ii) transfer of large amount of data between HMRC and other external government bodies had become a routine organisational practice.

This example points out clearly the fact that organisational information security cannot be contained only within technical specifications and solutions. It shows that many aspects of compliance with security policies are dependent on cultural interpretation of enforcement and on the interplay between explicit norms (policies) and implicit norms. Also, the HMRC case indicates that rituals and practices can conflict with the security norms. From a security perspective, an

organisation consist of a cultural system and a socio-structural system. The cultural system includes values, myths and ideologies that influence perceptions of security and security practice. Whereas, the socio-structural system consist in the institutional elements that are used to deploy security which include the functions, the policies, the procedures and the processes (Pieters and Coles-Kemp, 2011). Therefore tools and methodologies that help to specify and analyse security requirements that capture these aspects are a necessary addition to security management processes. In our opinion, taking a normative perspective to this problem would proffer helpful solution.

1.2 Security Goals

We have stated earlier that the general concerns about computer security comprise of confidentiality (secrecy), integrity, and availability. Confidentiality implies that certain subjects[1] of the system should not have access to certain information. This concerns the question of who has *permission* to know what information. Integrity is bothered about the protection of certain information from improper processing by some subjects such that its properties of usefulness and accessibility to other subjects are preserved. This can be viewed as a requirement of *obligation* on certain subjects to know certain information, consequently, the information must be protected before and during transmission to those subjects. Availability is concerned with the ability to access information when it is needed, that is while it is still useful. We can see that much security is based on the notion of *knowledge*, either what a subject is permitted to know or what a subject is obligated to know (Glasgow et al., 1992). For instance we can intuitively describe confidentiality in terms of what information a subject has permission to know. This therefore provides the opportunity to apply logic as a vehicle for reasoning about security, since logic allows us to reason about knowledge.

In contrast with the notion of confidentiality, integrity is concerned with the maintenance of consistency, accuracy, and trustworthiness of data over its entire

[1]We use the term subject here loosely to mean various actors that interact with the system, particularly with interest in the information available to the system. This will include human and non-human actors.

life cycle (Sabelfeld and Myers, 2003). Data must not be changed in transit, and steps must be taken to ensure that data cannot be altered by unauthorized people (for instance, in a breach of confidentiality). For example, a company's database system requires high integrity. This means that for instance, every transaction that got started has to be completed either with a *commit transaction* or a *roll-back transaction* in the event that the transaction fails to complete. As a result, accurate information can be accessed from the database at any point in time. There is no notion of secrecy here, but rather the question of accuracy of, and accessibility to, the stored information. Our understanding here therefore is that much of what integrity involves can be expressed using the *obligation* to know certain information or to perform certain actions. Integrity seems to involve some notion of time which can be in terms of ticking of a clock, interval between events, or deadlines. For example, using the database example, the begin transaction event is obliged to trigger a commit transaction event when completed, otherwise a roll-back event is triggered.

The third general security concern is that of availability. This can also be expressed in terms of obligation since it also involves a notion of time. Still using the database illustration, it is expected that information is accessible from the database when needed.

Following the discussions so far on the CIA security concerns, we approach the reasoning about security requirements associated with these concerns in terms of *permissions* and *obligations*. The formal reasoning tool that allows for the inclusion of these notions is found in modal logic, particularly temporal logic which has been applied to reasoning about computer security related issues such as security policy specification (Glasgow et al., 1992; Minsky and Lockman, 1985; Jones and Sergot, 1992) and security protocols verification (Burrows et al., 1990). However our position differs from these works in the sense that we are looking at security from a socio-technical perspective. In this case we are very much concerned about the interactions that occur between the system actors and how these interactions may affect the security of the system or the organisation. Hence our adoption of the normative systems perspective which also provides for the logic of actions, especially the notion of power in addition to the notions of *permission* and *obligation.*

In this work, we propose a computational approach to the verification and validation of elicited security requirements. The methodology is an institutional framework (Cliffe, 2007) which provides a mechanism to capture and reason about "correct" and "incorrect" behaviour within a certain context, which in this case is security. Based on logic programming, but inspired by deontic logic, the framework monitors the *permissions*, *empowerment*, and *obligations* of participants and generates violations when a security breach occurs. Information on the effects of participants' actions are stored in the state of the framework as facts. The "little" facts collected about events/actions triggered by participants over time may eventually lead to "big" facts that reveal vital information about a participant's behaviour with respect to the preservation of the system security. Using misuse case analysis for the initial elicitation of security requirements, our solution provides a means for rigorous testing of the security requirements elicited. The combination of our approach with the well-established misuse case approach provides a tool that can be used in determining the adequacy of a system's security requirements at design time. Implementation is achieved using an action language Inst*AL* (Cliffe, 2007) which is based on the semantics of answer set programming (ASP) (Baral, 2003).

1.3 Motivation and Scope

This research is motivated by the fact that the handling of organisational assets – primarily information – and processes is not entirely technical but also "social" in the sense that humans are intrinsically involved and interact in some social form in order to achieve organisational goals. This fact is well captured by the notion of socio-technical systems (STSs) which is largely credited to the action research project undertaken by Trist and Bamforth of the Tavistock Institute of Human Relations on British coal mines in which they assumed that organisations consist of the relation between human systems and non-human systems (Trist and Bamforth, 1951; Trist, 1981). Here the human and technical elements are taken to interface with each other in a typical industrial production system. For such a socio-technical system, the total system structure is composed of (Cooper and Foster, 1971): (i) the total set of *individuals*, their activities, and their interrelationships describing the *social system* of the factory; (ii) the total set of

machines and their displacement which describes the *plant layout* of the factory; and (iii) the total set of manufacturing *processes* and their interrelationships describing the *manufacturing system* of the factory. The interaction of these "social" and "technical" systems constitutes the socio-technical system.

Systems today are socio-technical in nature, for they consist of an interplay of social actors (human and organisations) and technical components (software and hardware) which interact with one another in order to achieve their objectives and requirements (Dalpiaz et al., 2013). For example, healthcare systems, smart cities, critical infrastructure protection systems, air traffic management control systems, etc. These systems have participants who are autonomous, heterogeneous and weakly controllable. As a result, a number of security issues are raised when these participants interact, especially when sensitive information exchange is involved in their interaction (Paja et al., 2013). The socio-technical nature of modern systems also makes it more vulnerable to the problem of social engineering (Winkler and Dealy, 1995; Thornburgh, 2004) which is the process of using non-technical means, including social interactions, to obtain critical information about a victim's system. This process can range from *i) impersonation* whereby an attacker pretends to be an authorised system user or official of an organisation and places a phone call in order to get the information needed, to *ii) scavenging* where the attacker goes through garbage from the target organisation (Thompson, 2013; Mitnick and Simon, 2002; Slatalla and Quittner, 1995). These scenarios describe the modern day security challenges in organisations.

It is not enough to consider technical mechanisms alone when dealing with the security problem in socio-technical systems because social aspects are also a concern. Today's organisational settings witness more and more interaction between human, technical and process components of the organisation as organisational goals are pursued. Therefore as organisations are not only composed of technical components, so also the security challenges that they face are not only technical but also include non-technical challenges made up of humans and processes. The inter-relationship between humans and technology (hardware and software) creates security vulnerabilities that could impact the organisation negatively. Such vulnerabilities are usually not easily understood early in the course of system design, since they are due to the "social" interaction between the ac-

tors in the organisation. Where this type of security vulnerability is thought of, the mitigation is usually specified in terms of high-level policy statements. The elicitation and analysis of requirements to tackle these types of security vulnerabilities is essential for ensuring that the security of information and processes is maintained in organisations. The work presented in this thesis is motivated by the need to provide methodologies and tools for the analysis of such security requirements.

1.4 Thesis Contribution

In this thesis, we present a computational approach to security requirements and compliance verification that is based on the application of formal reasoning techniques. Existing security requirements elicitation techniques attempt analysis by manual means. This is quite restricted due to the limitation of humans in being able to reason through the many possibilities of the kind of events that could be triggered by actors and the consequences of such actions on the security state of organisational assets – particularly information. Also the existing methods approach the problem of information security in organisations from a technical perspective and use deductive reasoning techniques for security policy analysis. These require complete specification of the system state in order to produce useful results. We emphasise in this thesis that security challenges that organisations face are not only technical but also social in the sense that user interactions and behaviours pose security challenges in the organisation. We therefore focus on the analysis of security requirements with this human factor in mind. Since the effectiveness of a security mechanism is largely dependent on how effective the elicitation of the system's security requirements was, it is important to be able to analyse the security requirement effectively before it is implemented. However, most of the existing approaches do not model the system behaviour in such a way as to support *a priori* analysis of requirements that are constraints on the runtime state of the system. One approach that attempted this (Bandara et al., 2003) used standard event calculus to model specifically authorisation and management policies. We propose a formalism that uses an action language to model processes/workflows and security requirements based on an institutional framework inspired by deontic logic. The reasoning mechanism is realised by

using answer set semantics.

The main contribution of this thesis is a computational approach for the analysis of security requirements using the institutional framework inspired by deontic logic and realised using Answer Set Programming. Further more:

- We demonstrate that the institutional framework can be used to develop a design time model of a business process in order to be able to reason about security requirements at design-time.
- We complement the misuse case approach for eliciting security requirements by providing a methodology for static analysis of the elicited security requirements.
- We review the challenge of managing information security requirements compliance and demonstrate how a business process can be analysed for compliance with such requirements.
- We develop a tool for generating specifications of a model in the action language instAL from UML-based models, thereby extending the original instAL development process. We also extend the query translation tool to allow for queries to be expressed in a more natural language manner.

1.5 Related Publications

In this section we present all the publications by the author which are related to this thesis, in accordance with Regulation 16.5 subsection k(ii) of the University of Bath Regulations for Students 2013/2014.

(Bibu, 2011) Bibu, G.D. Security in the context of multi-agent systems. In *The 10th International Conference on Autonomous Agents and Multiagent Systems – Volume 3,* AAMAS '11, pages 1339-1340, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.

This paper (an extended abstract) presents our initial exploration of the problem space and thoughts on our solution approach.

(Bibu and Padget, 2011) Bibu, G. and Padget J. Security engineering for

multi-agent systems: A normative approach (Poster). In *The 10th International Conference on Autonomous Agents and Multiagent Systems* AAMAS '11, Taipei International Covention Center (TICC) in Taipei, Taiwan, May 2011. International Foundation for Autonomous Agents and Multiagent Systems.

This poster presented the overview of the problem of security as including people, processes, and technology. We presented our solution framework which includes the integration of the institutional (normative) framework.

(Bibu et al., 2012b) Bibu, G.D., Yoshioka, N., and Padget, J. A. Security requirements analysis and validation using misuse case and institutional framework. Technical Report No. GRACE-TR 2012-2. GRACE Center of National Institute of Informatics, 2012. Tokyo, Japan. http://grace-center.jp/rsc_tr-html?lang=en.

This is the report of our work on analysing security requirements with misuse cases and the institutional framework.

(Bibu et al., 2012a) Bibu, G., Yoshioka, N., and Padget, J. System security requirements analysis with answer set programming. In *Requirements Engineering for Systems, Services and Systems-of-Systems (RES4), 2012 IEEE Second Workshop on,* pages 10-13, 2012.

In this paper we presented the implementation of our solution approach with results showing the analysis of security requirements.

1.6 Thesis Outline

Following this introduction, the remaining of the thesis is presented in Seven Chapters as follows:

In Chapter 2 (page 20) we present the review of the security requirements landscape as it relates to this thesis. We look at the traditional approaches to security requirements elicitation and analysis. We look at misuse cases and mal-activity diagrams as approaches for elicitation of security requirements and observe that these approaches do not present any computational

means of analysing elicited security requirements. We review and present use cases and scenarios in analysis of security requirements. Multi-agent system approaches to security requirements analysis are also presented. We present the various strengths and weaknesses in these approaches. We also introduce the concept of socio-technical systems and relate it to organisational information security in which the research presented in this thesis is situated.

Chapter 3 (page 33) then presents the approaches for modelling organisations with electronic institutions. This is to further lay the foundations on which the research is based. We review the existing approaches based on the two categories presented in literature namely: Organisational approaches and Institutional approaches. This review helps us to identify the suitability and the background of the approach we are adopting for the work presented in this thesis. We then proceed to introduce the Institutional Framework approach. We present the various aspects of the framework that are relevant to this thesis. We also present the formal definition of the Institutional Framework. This forms the basis for the next chapter which deals with the computational part of this work.

Chapter 4 (page 63) presents approaches for reasoning in computational logic. We start by introducing the Answer Set Programming (ASP) paradigm and its underlying logical formalism *AnsProlog*. We explain its syntax and semantics and compared ASP with other reasoning approaches. We then proceed to discuss the translation of the Institutional Framework into ASP, describing the translation of each component of the Institutional Framework. Finally we introduce the action language InstAL and describe how institutions can be specified in InstAL.

In Chapter 5 (page 107) we introduce the computational verification framework (CVF) for security requirements. We present an overview of the framework and we use a case study from the medical domain to show how the concept works. In generalising our approach we identify and present issues that concern the usability, completeness, and validation of the framework.

Chapter 6 (page 130) presents the tools developed and extended in order to

accomplish the research goals presented in this thesis. These include the XML2Inst*AL* translator which translates UML-based activity diagrams into Inst*AL* equivalent specifications. We present a formalisation for UML Activity Diagram semantics which guide the translation of our UML-based models to equivalent Inst*AL* specification. We also present some guidelines on reasoning with our framework through querying. The introduction of these tools also extends the original Inst*AL* reasoning framework in ways that makes its use easier. The use of these tools are presented with the results of analysis for which they were used.

In Chapter 7 (page 164) we present the analysis of verification of properties. Using appropriate scenarios for each of the security properties illustrated, we show how security properties may be analysis by querying the answer sets produced from the computational models of the scenarios in which the security property applies. We also present the verification of security requirements compliance. The results of the analysis are presented and discussed.

Chapter 8 (page 184) presents the summary of the thesis contribution and the future work that can follow from the work presented in this thesis. We also highlight some of the limitations of our approach in Section 8.3 on page 187.

Chapter 2

Security Requirements Engineering: The Landscape

2.1 Introduction

The development of modern Information Systems (IS) is faced with the challenge of security among other challenges such as performance and usability of such systems. This has drawn the attention of researchers to the topic of Security Engineering (SE). Security Engineering is defined as a discipline which explores how to build systems which are reliable in the face of malice, errors or mischief (Mouratidis et al., 2004). This encompasses a set of methods, techniques and tools responsible for eliciting, specifying, and analysing the requirements for protecting the resources of an IS to ensure that the security goals of information availability, confidentiality, and integrity are preserved. Security of IS has therefore progressively become a broad field of research.

A security requirement is a requirement which describes that part of a system shall be secure, or a property which, if violated may threaten the security of a system. Such requirements are derived from the traditional security goals of confidentiality, integrity, and availability. For example, an integrity security requirement can be stated as "The system shall prevent the unauthorized modification of data collected from customers".

The requirements engineering community has distinguished between two kinds of requirements: *Functional requirements* and *non-functional requirements*. There

is a broad consensus on the definition of *functional requirements*. Here, the emphasis is on functions, hence functional requirements are regarded as system requirements that specify functions that a system must be able to perform, what a system should do, and what the products must do (IEEE, 1990; Robertson and Robertson, 2006). Wiegers and Jacobson et al. presented more coherent definitions as follows: "A statement of a piece of required functionality or a behaviour that a system will exhibit under specific conditions." (Wiegers, 2003); "A requirement that specifies an action that a system must be able to perform, without considering physical constraints; a requirement that specifies input/output behaviour of a system." (Jacobson et al., 1999).

However, there is no harmonised definition for non-functional requirements. Glinz (2007) presents a summary of some of the definitions available in the community and it is noted that all definitions build on the terms which include property, attribute, quality, constraint, and performance. Non-functional requirements express constraints or conditions that need to be satisfied by functional requirements and design solutions. In other words, they define the overall qualities or attributes of the resulting system which are not related to the functionality of the system. It is common today to find non-functional requirements described as Quality of Service requirements, performance constraints, or "ilities" (reliability, extensibility, usability, availability etc). Therefore in contrast to functional requirements, which define what the system must do in terms of transforming inputs into outputs, non-functional requirements define non-behavioural attributes of a system which constrain the way in which the system must behave (Mirakhorli and Cleland-Huang, 2012).

Security requirements are generally considered non-functional requirements since they mostly concern what must not happen. They are therefore seen as restrictions or constraints on system services (Kotonya and Sommerville, 1998; Rushby, 2001; Mouratidis et al., 2003). Firesmith (2003) presents security requirement as a quality requirement specifying a required amount of security in terms of a minimum level system-specific criterion that is necessary to meet one or more security policies. This appears also as a form of constraint on the system.

In a security context a *threat* is seen as the potential for abuse of assets. A threat

is characterised in terms of an *attacker*, a presumed attack method, any vulnerabilities that are exploited by the attack, and the asset under attack. An *attack* is a sequence of events resulting in a threatening phenomenon. An attacker is considered a malicious actor, not necessarily a human, causing an attack. The Common Criteria for Information Technology Security Evaluation (commonly called Common Criteria) (Criteria, 2012) defines a vulnerability as the condition of a system which can be exploited by an attacker to cause a security violation.

In Figure 2-1 we present an overview of security requirements in relation to the other stakeholders in the security of a system. Central to any security endeavour in an organisation is the preservation of *assets* that are considered valuable to the organisation. These are the targets of any attacker, whether from within the organisation or from outside through the violation of security mechanisms implemented to protect the assets. For instance in a healthcare system, patients' medical record would be an asset that ought to be protected. These assets determine the security goals that the organisation's security efforts would be targeted at. Traditionally, security goals for information systems are classified into (i) *confidentiality* goals which are concerned with the protection of information assets against unauthorised access; (ii) *integrity* goals are bothered with the protection of assets from unauthorised alteration and corruption, and (iii) *availability* goals which are focused on the prevention of unauthorised degradation of the accessibility of assets. Still using the patients' medical records example, the security goal for a patient's health records would be primarily that of confidentiality. The determination of the security goals leads to the elicitation of the security requirements (constraints) which would mitigate the vulnerabilities in the system towards the security goal. A *Security requirement* is often considered a constraint on a functional requirement which ensures that a particular security violation cannot happen (Moffett and Nuseibeh, 2003). Therefore information security is achieved by introducing and implementing such constraints that satisfy a system's security requirements and providing protection to the organisation's information system and assets by inhibiting the attacker.

A number of approaches for security requirements have been proposed by researchers. We do not intend to present an exhaustive survey of these approaches in this thesis. However we highlight some of the approaches that are relevant to

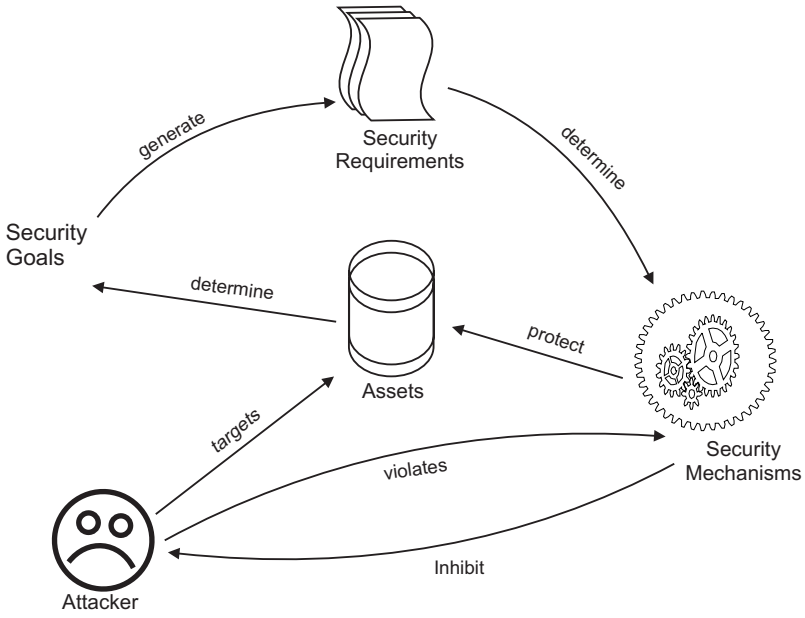


Figure 2-1: How Security Requirements relate to assets, attacker and security mechanisms

our work in terms of modelling language (formal or informal), perspective (organisational, Attacker oriented, System oriented), automated analysis tool, and socio-technical perspective.

2.2 Security Requirements Engineering - Traditional Approaches

As more information systems get connected and organisations' dependence on these systems increases, security threats that compromise these technologies are also growing. This is because the system witnesses more interactions between the actors in the system, thereby increasing the amount of information that flows within the system. Organisations are now more vulnerable to security problems than ever (Mitnick and Simon, 2002; PWC, 2012). Recent surveys on security breaches in organisations reveal that risky behaviours of users, including senior management employees, pose significant threats to information security in organisations (Friedberg, 2013; Green, 2014; Hawes, 2014). The challenge of managing employees' security behaviour has been echoed by security experts in organisations such as the Dow Chemical security manager Theresa Jones who was quoted in Johnson and Goetz (2007): "My biggest challenge is changing

behaviour. If I could change the behaviour of our Dow workforce, then I would think I had solved the problem".

Companies, in an effort to address security problems, spend millions of dollars on technical security solutions such as firewalls, encryption and secure access devices. These fail to provide the desired security because none of these measures address one of the weaker links in the security chain – humans and their behaviours (Poulsen, 2000; Beautement et al., 2008). The implication of this perspective on security is that the traditional approach to addressing security as only a technical problem would only yield security solutions that would not address the security challenges posed by the behaviours of humans. Many techniques and approaches have been proposed to specify security requirements and the Requirements Engineering community has acknowledged the importance of considering security since the early stages of software development (Lee et al., 2002; Pauli and Xu, 2005). We review some of the approaches used in the requirements engineering community for capturing and expressing requirements as follows;

2.2.1 Use Cases and Scenarios

A use case is the set of interactions that a user or other system component has with a system in order to achieve a goal. The term *Actor* describes the person or system associated with a goal, implying that actors are driven by goals. Use cases (Cockburn, 2001; Rumbaugh, 1994) are popular for determining, communicating, specifying, and documenting requirements. They are used to describe interactions involving systems, actors and their environments. Each use case specifies a sequence of actions that a system can perform, including variants, interacting with actors of the system. Ever since the adoption of use cases by software development approaches such as the Unified Process (Jacobson et al., 1999), use cases have become one of the favourite approaches for requirements capture. Several use cases are usually needed to completely describe a system's requirements. In terms of UML syntax and semantics, relationships between use cases and between actors and use cases are represented in a use case diagram. Typical relationships between use cases in the UML are *include* and *extend* relationships.

Some of the problems identified with use-case-based approaches to requirements engineering include such things as over-simplification of assumptions about the problem domain and premature design decisions (Arlow, 1998; Lilly, 1999). While use cases are suitable for most functional requirements, they may lead to neglect of extra-functional requirements, such as security requirements (Alexander, 2003). As a result of these, we do not find use cases too useful for the purpose of this thesis. We rather focus on the use of scenarios which describe interactions between systems and actors in a sequence that lead to the desired goal. A scenario may be defined as a sequence of triggers, system reactions, waiting delays, guard realizations and assertions. Triggers are operations from actors in the environment, while system reactions are operations of the system. A waiting delay specifies a point in a scenario where a certain amount of time passes without any trigger or system reaction. A guard realization is a condition set to hold at a certain point in time. The following example illustrates how a use case differs from a scenario:

`Use Case: User authenticates with ID and password`

Possible Scenarios:

`1. ID is recognized, password is correct.`
`2. ID is recognized, password is incorrect.`
`3. ID is not recognized.`

Scenarios present possible ways to use a system to accomplish some desired function, they are therefore use-oriented. Scenario-based approaches have attracted increasing interest among requirements engineers and the literature abounds on scenario methods, models, and notations (Carroll (1995); Alexander and Maiden (2004); Dzida and Freitag (1998); Hertzum (2003)). Scenarios have also become popular in other fields, notably human-computer interaction and strategic planning.

2.2.2 Misuse Case and Mal(icious)-activity Diagrams

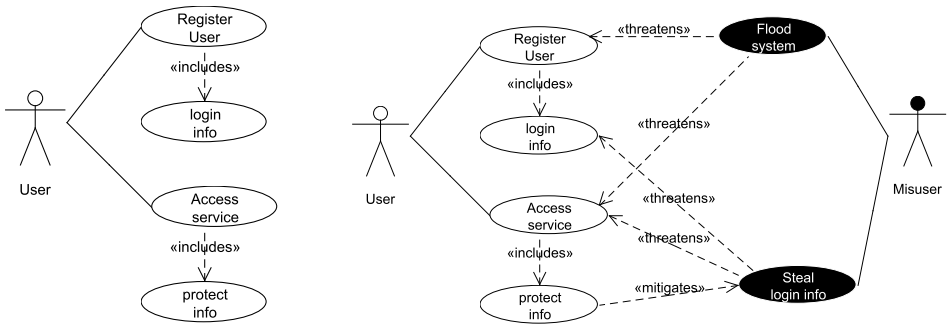
Misuse Case

Techniques used for modelling functional security requirements at the early stage of system development cycle include the misuse cases and mal-activity diagrams. Early consideration of security requirements allows system designers to equip their systems with security mechanisms built within system design rather than relying on external defensive mechanisms. A misuse case is a sequence of actions and interactions between actors (system and/or users) such that if allowed to complete would cause harm to some stakeholder. The user in this case is called a *misuser* who initiates the misuse case intentionally or inadvertently. In a model-driven engineering process, misuse cases are expected to drive the construction of mal-activity diagrams.

Misuse case modelling emerged from the more popular UML *use case* functional requirement specification technique and has become a promising technique in the past decade for the elicitation and modelling of functional security requirements (Sindre and Opdahl, 2005; Alexander, 2003; Pauli and Xu, 2005). A misuse case is a use case from the point of view of a hostile actor. Hence structurally, a misuse case diagram can simply be said to be an ordinary use case diagram annotated with misuse cases and misusers. However functionally, use cases define the intended usage scenarios of a system by its intended entities whereas misuse cases define improper usage scenarios of a system by misuser[1] entities that may lead to harmful consequences irrespective of whether these scenarios were performed intentionally or unintentionally (El-Attar, 2012a). Figure 2-2 shows a simple illustration of use case (2-2a) and misuse case (2-2b) diagrams. The misuser and the misuse cases are shown in inverted colour and the interactions between the misuse cases and use cases are denoted with `<<threatens>>` and `<<mitigates>>` depending on the kind of effect they have on each other. The rule governing the description of the interaction between any use case and misuse case is presented in Table 2.1 on the next page.

Misuse case are described using natural language so stakeholders are able to

[1]We use the term *Misuser* in this thesis in a generic sense to mean any unintended user or unauthorised user



(a) A Use Case (b) A Misuse Case

Figure 2-2: Illustration of Use Case and Misuse Case Diagrams of a Simple Scenario

		Target case type	
		Use	Misuse
Source case type	Use	Includes	Mitigates
	Misuse	Threatens	Includes

Table 2.1: The Rules for Interaction between use and misuse cases

understand it easily and give feedback since misuse case modelling by nature inherits key features from use case modelling (El-Attar, 2012a). Also, misuse cases describe security requirements as scenarios, that is a set of operational action sequences. In an industrial survey presented in Weidenhaupt et al. (1998) which spanned from small applications to large application, scenarios have been shown to be an effective technique for determining and validating requirements. Finally, since misuse case modelling is performed during the requirements engineering phase, security concerns are in turn considered prior to the initial architectural design of a system. With misuse cases, focus can be put on security by describing security threats and then requirements, without going into design. However misuse cases just like use cases have their weaknesses that make them unsuitable for analysis of security requirements. First a sequence of actions and actors (misusers) cannot be identified from misuse case. As such identifiable sequence of actions cannot be exploited for analysis. Secondly the lack of unified formal semantics is a problem for automatic translation and analysis of misuse case models. Misuse cases are best for capturing threats and attacks just at the system boundary (whether that boundary delineates a fully automated system

or a human organisation). They are less suitable for capturing attacks that take place either inside or outside the system (Sindre, 2007). Take for instance an insider who performs fraudulent actions within an organisation. Assume the fraud requires the collusion of several insiders in the organisation who may perform their actions at different times and probably as part of several normal use cases. Misuse cases would not be useful in capturing such an attack since misuse cases would only give an overview of the normal functions wanted in a system, as well as the threats posed by attackers, and in part which threats are related to which normal function. They are not able to show sequences of activities hence not able to show where a certain malicious activity might fit into the organisational business process. This implies that if only misuse cases are produced, there will remain a large gap between the analysis and design phases with respect to developing the necessary security mechanisms. Misuse case models therefore provide an excellent starting point to model security requirements, but they are insufficient on their own. The limitations of misuse cases are overcome by mal-activity diagrams (Sindre, 2007).

Mal-activity Diagrams

Just as misuse case diagrams are inspired by use cases, mal-activity diagrams are inspire by UML activity diagrams. Mal-activity diagrams complement activity diagrams by adding malicious users/actors and their activities in a scenario. Therefore similar to the relation between misuse cases and use cases, mal-activity diagrams use the same syntax and semantics as UML activity diagrams. Malicious activities are depicted as normal activity icons but with inverted colours, malicious actors as misusers but with white text on a black background with malicious actors (misusers) having designated swim lanes. Where there are malicious decisions, these are depicted as ordinary decision diamonds but with the black filling. Mal-activity diagrams model security requirements more precisely while providing initial guidance towards a detailed system design. In effect, mal-activity diagrams help bridging the gap between the analysis and design phases with respect to developing the necessary security mechanisms.

In our work we make use of activity diagrams to capture scenarios since we are interested in identifying actors and their actions in a defined business process.

Our activity diagrams are then translated into an InstAL model based on the Institutional Framework translation described in Section 4.5 on page 88.

2.2.3 Abuse Cases and Anti-models

The Abuse Case (McDermott and Fox, 1999) is an adaptation of the use case modelling technique (Firesmith, 1999) which characterises the way a system is expected to be used. While a use case is a specification of a kind of complete interaction between a system and one or more actors, an abuse case specifies a type of interaction between a system and one or more actors, where the results of the interactions are negative/harmful. It includes a range of security concerns that might be abused, as well as a description of the harm that might be caused.

Anti-models were proposed in van Lamsweerde (2004) to deal with security engineering at the application layer. The anti-model, which is built to capture attackers, includes vulnerabilities and capabilities needed to achieve the anti-goals (the attacker's goal) of the security goals that are endangered. Anti-models are designed to lead to the generation of more subtle threats and the derivation of more robust security requirements as anticipated countermeasures to such threats. Anti-goals are refined in threat trees, whose leaf nodes represent either vulnerabilities observable by the attacker or anti-requirements implementable by the attacker. This approach does not focus on the social interaction among actors. It is therefore not suitable for our approach.

A goal-oriented methodology based on i^* modelling language (Yu et al., 2011) was presented in Liu et al. (2003) to deal with security and privacy requirements. Security dimensions are modelled as soft-goals, and security requirements analysis is performed to verify whether the system is secure. Analysis identifies potential system attackers/abusers, vulnerabilities (propagated along dependency links), thereby suggesting countermeasures. Their solution falls short when considering security issues through the later phases of the development process.

2.3 Security Requirements Engineering: Multi-agent System Approaches

2.3.1 Secure *i**

This modelling framework extends and refines the i^* modelling language (Yu et al., 2011) for software requirements engineering. Proposed for the design of socio-technical systems, the framework consist of primitives for capturing organisational security aspects. These primitives include *actor*, *goal*, *task*, *resource*, and *social dependencies between actors*. The framework is enhanced with the notions of permission, delegation, trust, and supervision which are at the centre of its security concerns. Delegation is used to model the transfer of rights between actors while trust is used to model the expectation of an actor about the behaviour of other actors (Zannone, 2009). Si* employs a number of modelling activities to express the design of secure STSs, each producing a diagram representing a view of the requirements model. These include; An *actor diagram* which describes domain stakeholders and system's actors. Concrete actors are explicitly modelled as *agents*. These could refer to human as well as software agents and organisations. Abstract characterisation of the behaviour of the social actor within some defined context is modelled as a *role*. An *agent* is therefore said to play a *role*. The concept of role hierarchies is also implemented using the concept of specialisation. Actors are described in terms of their *objectives*, *entitlements*, and *capabilities*. Objectives could be goals intended to be achieved, tasks to be executed, or resources to be acquired by the actor. Entitlements consist of goals, tasks, and resources that the actor has the permission to achieve, execute, and furnish respectively. Capabilities are the actor's ability to achieve, execute, or furnish goals, tasks, or resources respectively.

The social relations among the stakeholders and system's actors are described in terms of the organisational structure and expectations about the behaviour of other actors. These social relations are modelled as *social diagram*. The concepts of supervision, trust of permission, and trust of execution are used together with the concepts of agent and role to build organisational hierarchies. Trust of execution is used to model the expectations one actor has on the competence and

dependability of another actor. In general, by trusting in execution, an actor is sure that the other actor will achieve the goal, perform the task, or furnish the resource. Trust of permission is used to model the expectations of one actor that another actor does not misuse a goal, a task, or a resource. By trusting in permission, an actor is sure that the (possibly) granted permission is properly used.

Interesting to us in this work is the way the notion of permission is used. It is used in the sense of actors granting each other authorisations. However, we use the notion of permissions to determine actions that would not be considered as violations of rules, hence would not cause a breach of security in the system.

2.3.2 Secure Tropos

Secure Tropos is an extension of Tropos, which is an information system development methodology, tailored to describe both the organisational environment of a system and the system itself. Although Tropos incorporates the i^* modelling framework (Yu et al., 2011), it does not handle security issues for the system under consideration. Secure Tropos (Mouratidis, 2011) extends Tropos by introducing concepts such as security constraint and secure dependency among others, in order to provide a systematic process that will guide the developer in considering security requirements throughout the phases of system development. Security requirements are expressed as security constraints, which should be satisfied together with the functional requirements. Potential threats and attacks are considered as well, to analyse and find the best way to overcome possible vulnerabilities. Analysis of security is done through a graphical representation where nodes represent the actors and the link between nodes represent the dependencies between the nodes. The major problem for us with this approach is the imposition of security constraints on dependencies which means that agents are constrained to act in particular predetermined ways. This leaves no room for autonomy of the agents in terms of behaviours. Humans have the ability to think and behave rationally, that is, to chose actions that maximises their expected utility and this demands autonomy (Wooldridge, 2000; Deci and Ryan, 1987). Imposition of constraints also means that violations are not expected, hence there would be nothing for us to analyse.

2.4 Summary of Chapter

Today, security can be broadly classified into internal threats and external threats. Most organisations appear to invest much in security technologies in readiness against external threats such as virus threats, hacker attacks, and infrastructure failures such as fires, terror attacks, distributed denial of service attacks, fraud attacks and Spam. The Ernst & Young's 2012 information security survey (ey1, 2012) shows that organisations have improved greatly on their information security capabilities. However, they are still behind in ensuring information security *within* their organisations, thereby creating an information security gap that grows ever larger. A lot of the challenges faced by the organisations are therefore due to the other category of security - internal threat. These are security threats that arise as a result of accidental or intentional security breaches done by employees through negligence, non-conformance to organisational information security policies, or misconduct on systems among several other possible misbehaviours. The human factor challenge in information security therefore starts with intentional or involuntary employee misconduct. This challenge is due to the fact that most often the perpetrators use unsophisticated methods to gain access to systems; Something as simple as an employee sharing his/her passwords or writing a hard-to-remember password and sticking it on the work desk, an employee leaving a confidential document openly on the desk, or a disgruntled employee having access to the emails even after leaving the organisation. All these could cause a security breach, and the possibilities are many (Surendran, 2005). Therefore, modelling and analysing the organisational environment (actors and processes) where the system will act is crucial for building secure systems. This allows designers to identify security mechanisms that can best protect the system, and their impacts on the system, since security breaches often occur at an organisational level, rather than a technical one (Anderson, 1993). In this thesis, we argue that organisational security of information systems can be enhanced by analysing the behaviour of the actors within the organisation. The analysis will help in the development and implementation of the appropriate security mechanisms that will mitigate some of the security vulnerabilities. This, we claim can be achieved by using agent-oriented modelling approach to model the organisation as a set of institutions.

Chapter 3

Modelling Organisations with Electronic Institutions

3.1 Introduction

The notion of *institution* is used in different contexts in everyday language. For example a given hospital is described as a "health care institution", or we talk about "marriage institution". These everyday uses and many other forms of institution have been studied and formalised by legal theorists, political scientists, economists, and philosophers (see Powell and DiMaggio (1991); Aoki (2001)). Some of the features presented in these conventional understandings include a distinction between *institutional facts*, for example x **owns** y, and *brute facts* for example x **has-possession-of** y, and the assumption that institutions involve regulations, norms, and conventions in which some approaches presented by North (1990) and Ostrom (1986) take institutions to be the conventions themselves, thereby drawing a clear distinction between institutions and organisations. Others take institutions to be organisations consisting of rules or norms, institutional objects and due processes or procedures, but still keep individuals out of the institution. As a result of these understandings and their formalisations, the agents community has used the notion of institutions to model and implement a variety of socio-technical systems serving the same purpose as conventional institutions. The aim is to enable and regulate the interaction among autonomous participants towards achieving some collective goals. In this case, interactions in the agent institution must comply with some conventions, rules, and norms

that apply to every participating agent. Regulations control interactions and are applicable to individual agents on the basis of what they do (i.e roles they adopt) rather than on the basis of who they are (Fornara et al., 2013).

The concept of institutions has its root in social sciences (North, 1990) and are regarded as the humanly devised constraints that shape human interaction, thereby reducing uncertainty as they provide structure to everyday interactions among humans. These institutions can be formal constraints such as rules and regulations formulated by humans, or informal constraints such as conventions, norms, and codes of behaviour (North, 1990). Institutions are regarded as the framework within which human interactions take place. They consist of constraints such as prohibitions and conditions under which certain actions are permitted by certain actors. However, in human interactions, it is a known fact that norms, whether formal rules or informal codes, are usually violated either deliberately or unintentionally. An essential part of an institution therefore is the detection of violations and its effect on the society.

In this work, the focus is not on human society in general but what we consider a subset of the society, an organisation which is defined as a social unit (or human grouping) constructed deliberately to pursue specific goals (Esteva et al., 2001). In addition to this we also consider an organisation as a unit which consist of a system of interacting human and technical actors constructed for specific goals. In this domain, there are constraints analogous to the institutions in a human society as discussed earlier, which specify what actions are acceptable and by what actors. There are also constraints that specify conditions under which certain actions can take place. These are known as information security policies (ISPs) which organisations provide with the aim of controlling behaviours by providing employees (actors) with guidelines on how to ensure information security while they utilize information systems in the course of performing their jobs (Whitman et al., 2001; Bulgurcu et al., 2010).

However, these actors are autonomous, heterogeneous, independent, unreliable, and liable. In the light of socio-technical systems, human actors have their own minds and would tend to behave in ways which are divergent from the expected pattern of behaviour. Also, things could go wrong with technical systems, such as

hardware failures and software bugs, causing them to exhibit behaviours which are inconsistent with the expected behaviours. These divergent human behaviours and failures in the technical components of a system could make the system vulnerable to security attacks which would not have been determined at system requirement and design time.

3.2 Organisations and Institutions

According to North (1990), institutions provide a structure for everyday life which guide human interactions. Institutions are the framework within which human interact as they define prohibitions, permissions and other conditions such as obligations for participants. In our opinion, the notions of *institution* and *organisation* are closely related. The essential distinction is that the institution is focused on what can be done, while organisations focus on who does it. The increased complexity of real-world applications, which is exacerbated by the internet has called for the need to incorporate organisational abstractions into computing systems. This would enable the connection between the cyber and the physical due to the potential for conceptual alignment thereby easing the design, development, and maintenance of computing systems. This calls for the use of electronic institutions which provide a computational analogue of human organisations in which agents (representing humans) play different organisational roles and interact with each other and possibly humans to accomplish individual and organisational goals (Sierra et al., 2004). There are several research groups that see collections of heterogeneous agents as agent societies, and thus try to bring solutions from human societies into distributed application scenarios with heterogeneous actors. In these approaches the aim is to model a given society or organisation by defining some kind of (social) structure that establishes the (accepted) relations among agents or roles. This is important for the specification and balance of autonomy and control. There are two main approaches (ALIVE, 2010):

1. organisational approaches which construct the social structure by means of defining roles stating the restrictions to be followed by the agents that enact such roles.

2. Institutional approaches which create the social structure by an accurate definition of the norms to be fulfilled by a given agent and the relations of deontic influence between agents.

3.3 Organisational Approaches

In human organisations, human beings define various forms of constraints that help guide behaviours and interactions in that organisation. These are seen as institutions (Sierra et al., 2004) which generally represent the rules of the game in a society, defining what individuals are forbidden and permitted to do and under what conditions. Human organisations and individuals must conform to the rules of institutions in order to receive legitimacy and support. Therefore, an organisation can be seen as a set of entities and their interactions, which are regulated by mechanisms of social order (institutions) and created by more or less autonomous actors to achieve common goals. According to Dignum (2004), organisations, as social systems, comprise a factual and a procedural dimension. The factual dimension consists of the observable behaviour of the organisation, that is, high level goals, inputs and outputs. The procedural dimension has to do with how this behaviour is obtained, that is, the division of labour into roles, the determination of authority lines and the establishment of communication links. Organisations are created to provide the means for coordination in order to achieve desired goals. The structuring of an organisation facilitates the flow of information within the organisation in order to reduce the uncertainty of decision making. The structure also should facilitate the integration of organisational behaviour across the different parts of the organisation for easy coordination (Brazier et al., 2012). This concept of organisation form the basis of our approach towards security requirements analysis for socio-technical systems. We would like to be able to determine the events and actors that are crucial to maintaining the security status of the organisation.

Organisational approaches aim at constructing social structures by means of roles and the relations and the restrictions over the agents that enact these roles. The literature abound on organisational approaches. We review some selected approaches which include OperA (Dignum, 2004; Quillinan et al., 2009), OMNI(Dignum et al., 2004), AGR (Ferber et al., 2003), Moise+ (Hannoun et al.,

2000; Hübner et al., 2002, 2007), and O-Mase (García-Ojeda et al., 2007).

3.3.1 OperA Framework

In OperA framework (Dignum, 2004), agents are seen as autonomous communicative entities that will enact societal role(s) as a means to realize their own goals according to their own internal aims and architecture. Interaction between agents is represented in such a way that: it is independent of the internal design of the agents; distinguishes organisational characteristics from agents' own goals; creates dynamic links between organisational design and agent populations; and allows for the adaptation of interaction patterns to the characteristics of specific populations. That is, the OperA model enables the specification of organisational policies, such as objectives and norms and at the same time allowing participants to have the freedom to act according to their own capabilities and demands. It assumes organisations as being open systems, as such, it does not include constructs for the specification of the actual agents, treating them as "black boxes" that commit to a specific (negotiable) interpretation of the organisational roles (ALIVE, 2010). OperA therefore meets the following requirements (adapted from Dignum (2004)):

- Internal autonomy requirement: The internal behaviour of the participating agents should be represented independently from the behaviour of the society.
- External autonomy requirement: The external behaviour of the participating agents (i.e., interaction with other agents or the environment) should be specified without completely fixing the interaction possibilities in advance.

The OperA agent society framework is presented in terms of three interrelated models viz;

i.) Organisational Model (OM)- which represents the organisational structure of the society, consisting of roles and interactions, as intended by the organisational stakeholders;
ii.) Social Model (SM)- in which the enactment of roles by agents is fixed in social contracts that describe the capabilities and responsibilities of the agent

within the society, that is the agreed way the agent will fulfil its role(s); and

iii.) Interaction Model (IM) - which describes the possible interaction between agents in a given agent population.

Although OperA is formally founded in deontic logic, it is not executable without an implementation language since it does not specify the internal behaviour of agents and only specifies the "what" and not the "how". An agent-based implementation language such as Brahms (Clancey et al., 1998) is needed to define the "how".

Okouya and Dignum (2008) presents OperettA as a graphical tool that supports the design, verification and simulation of OperA models. It ensures consistency between different design parts, provides a formal specification of the organisation model, and is prepared to generate a simulation of the application domain. The OperettA prototype is implemented using Meta-Edit+, a generic customizable model driven software development environment suitable for prototyping. The prototype incorporates Racer DL reasoning system (Haarslev and Müller, 2001), SWI-prolog interpreter (Wielemaker et al., 2012), MCMAS model checker (Lomuscio and Raimondi, 2006) and BRAHMS (Clancey et al., 1998) as a possible simulation environment.

3.3.2 OMNI

The organisational Model for Normative Institutions (OMNI) (Dignum et al., 2004) integrates the normative concepts of HarmonIA (Vázquez-Salceda, 2003) with the organisational concepts of OperA (Dignum, 2004). OMNI is an integrated framework that can be used to model a whole range of MAS, from closed systems with fixed participants and interaction protocols, to open, flexible systems that allow and adapt to the participation of heterogeneous agents with different agendas. This is because:

- It specifies global goals of the system independently from those of the specific agents that populate the system.
- Both the norms that regulate interaction between agents, as well as the contextual meaning of those interactions are modelled.

OMNI is composed of three dimensions that describe different characterizations of the environment:

- Normative dimension, which models all normative and regulatory aspects of the agent organisation.
- organisational dimension, which models the social structure, the roles, the intended interactions and the role enactment by agents.
- Ontological dimension, which defines the ontologies for communication and also the ontologies of the concepts appearing in both the normative and organisational dimensions.

The framework is further organized into three levels of abstraction:

- The Abstract Level: where the statutes of the organisation to be modelled are defined in a high level of abstraction. This step is similar to a first step in the requirement analysis. It also contains the definition of terms that are generic for any organisation (that is, that are in contextual) and the ontology of the model itself.
- The Concrete Level: where all the analysis and design process is carried on, starting from the abstract values defined in the previous level, refining their meaning in terms of norms and rules, roles, landmarks and concrete ontological concepts. In order to check norms and act on possible violations of the norms by the agents within an organisation, on the normative dimension, abstract norms have to be translated into actions and concepts that can be handled within such organisation. The organisational dimension specifies the means to achieve the objectives identified in the abstract level as an organisational Model. The content aspects of communication, or domain knowledge, are specified by Domain Ontologies and Generic Communication Acts define the interactions languages used in the organisational Model.
- The Implementation Level: where the design in the Normative and organisational dimensions is implemented in a given multi-agent architecture. Describes the implementation of the design in a given multi-agent

architecture, including the mechanisms for role enactment and for norm enforcement. The normative dimension provides both the low-level protocols and the related rules that enable agents to comply with organisational norms. OMNI assumes that individual agents are designed independently from the society to model goals and capabilities of a given entity. Agent populations of the organisational model are described in the Social Model in terms of commitments regulating the enactment of roles by individual agents. Depending of the specific agents that will join the organisation, several populations are possible for each organisational model.

The modular structure of OMNI facilitates the adaptation of the framework to different types of domains. In those domains with none or small normative components, design is guided by the organisational Dimension, while in highly regulated domains the Normative Dimension is more prominent and therefore guides the design. However, there is no known tool yet for the implementation of this framework.

3.3.3 MOISE+

Adopting a different perspective, Hubner et al. (see Hannoun et al. (2000); Hübner et al. (2002, 2007)) presented Moise+ which is designed as an organisational model that explicitly distinguishes three dimensions in the modelling of an organisation namely structural, functional and deontic dimensions:

1. The structural dimension defines the agent's relationship through the notions of roles, groups and links by specifying three levels thus:

   i. the behaviours that an agent is responsible for when it adopts a role (individual level),

   ii. the acquaintance, communication, and authority links between roles (social level), and

   iii. the aggregation of roles in groups (collective level).

   In Moise+ model, the adoption of roles is constrained by a compatibility relation between roles. An agent can play two or more roles only if they

are compatible.

2. The functional dimension describes how a multi-agent system usually achieve its global (organisational) goals stating how these goals are decomposed (by plans) and distributed to the agents (by missions - a set of coherent goals that an agent can achieve).

3. The deontic dimension addresses the autonomy of the agents by stating explicitly what is permitted and obligated in the organisation. The corresponding specification describes the roles' permissions and obligations for missions. A permission permission(ρ,$\mathtt{m}$) states that an agent playing the role ρ is allowed to commit to the mission m. Furthermore, an obligation obligation(ρ,$\mathtt{m}$) states that an agent playing ρ ought to commit to $\mathtt{m}$.

The Moise+ organisational model is an attempt to join these three dimensions into an unified model where the first two dimensions can be specified almost independently of each other and afterwards properly linked by the deontic dimension. This linkage allows the MAS to change structure without changing the functioning, and vice versa, the system only needs to adjust its deontic relation (Hübner et al., 2005). Moise+ has been extended with a J-Moise+ agent implementation level which is an extension of the AgentSpeak Jason features (Bordini et al., 2007). Additionally, S-Moise+ is an organisational middleware that connects the organisational model to the implementation level. It provides agents with the current state of the organisation and allows agents to change the organisation entity (OE) and specification (Hübner et al., 2005). S-Moise+ is an open source implementation of an organisational middleware that follows the Moise+ model. This middleware is the interface between the agents and the overall system, providing access to the communication layer , information about the current state of the organisation (created groups, schemes, roles assignments, etc.), and allowing the agents to change the organisation entity and specification. Of course these changes are constrained to ensure that the agents respect the organisational specification. S-Moise+ has two main components: an OrgBox API that agents use to access the organisational layer and a special agent called OrgManager. This agent has the current state of the OE and maintains it consistently. The OrgManager receives messages from the agents' OrgBox asking for changes in

the OE state (e.g. role adoption, group creation, mission commitment). This OrgManager changes the OE only if it does not violate an organisational constraint. For example, if an agent wants to adopt a role $\rho 1$ but it already has a role $\rho 2$ and these two roles are not compatible, the adoption of $\rho 1$ must be denied.

3.3.4 AGR

Ferber et al. (2003) noted that except in very small organisations, organisations are structured as aggregates of several partitions, sometimes called groups or communities, contexts, departments, services, etc. and each partition may itself be decomposed into sub-partitions. They therefore proposed the AGR (agent/-group/role) model, based on three primitive concepts, Agent, Group and Role that are structurally connected in an organisation and cannot be defined by other primitives. An Agent is seen as an active, communicating entity playing roles within groups. An agent may hold multiple roles, and may be member of several groups. An important characteristic of the AGR model, is that no constraints are placed upon the architecture of an agent or about its mental capabilities. Thus, an agent may be as reactive as an ant, or as clever as a human. A group is a set of agents sharing some common characteristics. A group is used as a context for a pattern of activities, and is used for partitioning organisations. Two agents may communicate if and only if they belong to the same group, but an agent may belong to several groups. This feature allows the definition of organisational structures. The Role is the abstract representation of a functional position of an agent in a group. An agent must play a role in a group, but an agent may play several roles. Roles are local to groups, and a role must be requested by an agent. A role may be played by several agents.

3.3.5 O-MaSE

García-Ojeda et al. (2007) attributed the lack of widespread industrial acceptance of the many processes for developing MAS to the lack of Computer Aided Software Engineering (CASE) tools that support the process of software design. They therefore presented the organisation-based Multi-agent System Engineering (O-MaSE) Process Framework, based on the Multi-agent System Engineering

(MaSE) methodology (DeLoach et al., 2001) with the goal of allowing process engineers to construct custom agent-oriented processes using a set of method fragments based on a common meta-model. The O-MaSE meta-model is based on an organisational approach which is composed of five entities: Goals, Roles, Agents, Domain Model, and Policies. A Goal defines the overall function of the organisation, while a Role defines a position within an organisation, whose behaviour is expected to achieve a particular goal or set of goals. Agents are human or artificial (hardware or software) entities that perceive their environment and can perform actions upon it. In order to perceive and to act in an environment, agents possess Capabilities, which define the percepts/actions the agents have at their disposal. Capabilities can be soft (i.e., algorithms or plans) or hard (i.e., hardware related actions). Plans capture algorithms that agents use to carry out specific tasks, while Actions allows agents to perceive or sense objects in the environment. This environment is modelled using the Domain Model, which defines the types of objects in the environment and the relations between them. Each organisation is governed by rules, which are formally captured as Policies. A Policy describes how an organisation may or not may behave in a particular situation.

3.4 Institutional approaches

All human societies use social constraints to regulate the relations among its members. These social constraints are some sort of conventions and rules which govern human societies by defining the interactions between members of the society. The conventions and rules either originate from the normal practice in the society or from laws that were developed by the society. They are either informal (for instance customs and traditions) or formally defined (for instance laws and regulations). The constraints of a society influence the actions of its members who adopt them, thereby creating patterns of expected behaviour. A collection of these social constraints are referred to as *institutions* (North, 1990). The integration of institutional constraints in a society allows every participant to act according to the norms or rules of the institution.

We present a review of relevant frameworks which have been proposed for the specification of organisations as multi-agent systems in the context of institu-

tions. These are the E-Institutions (Esteva et al., 2002; Sierra et al., 2004; Esteva et al., 2004) and HarmonIA (Vázquez-Salceda and Dignum, 2003; Vázquez-Salceda, 2003).

3.4.1 E-Institutions

This is a framework for electronic institutions that was first presented in Esteva et al. (2000). The framework consists of a formal specification of the institution, a tool for editing the specification called ISLANDER, and a middleware called AMELI for executing the system based on the specifications (Esteva et al., 2002, 2004). The concept of e-institutions include formal semantics for the core notions of the framework. The core notions of interest to us include agents and roles, scenes, and normative rules.

Agents are defined as the players in the electronic institution while roles are taken to be standardised pattern of behaviour. Agents are required to adopt some roles and agents are allowed to perform actions associated with the adopted roles. This is similar to our framework in which roles, actors, and actions are defined and actions are associated with actors adopting roles. Scenes describe the articulation of agent interaction through well defined communication protocols. This could be seen as the possible dialogue agents could have. The *a priori* definition of the interaction protocol makes e-institutions unsuitable for our work. This is because the careful definition of the protocols means that agents cannot interact outside the defined protocols. We are looking at a situation where our actors can freely interact in which case, some of the interactions, such as violations, could lead to vulnerabilities to the system. Normative rules are presented in the context of limitations which are placed on agents as a result of consequences of the agent's previous actions. This is expressed using the deontic notion of obligation. The schema of the norms are therefore built around the notion of obligation.

Drawbacks of e-institutions include the fact that most of the norms that define the workings of the institution are flattened and incorporated into the scene and inter-scene structure, without any explicit representation. Also, the implementation has the problem that the governors restrict the agents to the protocol that was defined on forehand, thus severely limiting the autonomy of the agents partic-

ipating. This decreases the flexibility and robustness of the system overall.

3.4.2 HarmonIA Framework

The HarmonIA framework is first introduced in Vázquez-Salceda and Dignum (2003) as a framework that defines a multilevel structure, from the most abstract level of a normative system to the final implementation of the organisation. It is composed of four levels of abstraction:

1. The Abstract Level: where the statutes of the organisation are defined in a high level of abstraction along with the first abstract norms.
2. The Concrete Level: where abstract norms are iteratively concretised into more concrete norms, and the policies of the organisation are also defined.
3. The Rule Level: where concrete norms and policies are fully refined, linking the norms with the ways to ensure them.
4. The Procedure Level: where all rules and policies are translated in a computationally efficient implementation easy to be used by agents.

The division of the system into these four levels aims to ease the transition from the very abstract statutes, norms and regulations to the very concrete protocols and procedures implemented in the system. It is specially suited for those complex, highly regulated domains where the behaviour of a real organisation or an e-organisation has to follow regulations that define restrictions at different levels of abstraction (Vázquez-Salceda, 2003).

In an attempt to solve part of the restrictive nature of the ISLANDER formalism, HarmonIA proposed the use of so-called *Police Agents*, that will be responsible for the enforcement of the norms. Such an implementation would allow the safety of the norms, while still allowing the agents (enough) autonomy to perform their tasks in manners that were not thought of at design, thus enabling them to handle unforeseen situations and adding a level of robustness to the system (ALIVE, 2010).

The ideas of Vazquez (Vázquez-Salceda, 2003) were further extended by Aldewereld (Aldewereld, 2009) who applied parts of the methodology to highly-regulated environments (environments governed by lots of complex norms). He identified

four important aspects of institutional implementations: 1) an ontology to allow communications between agents, and to express the meaning of the concepts used in the norms; 2) an (explicit) normative description of the domain, specifying the allowed interactions in the institution, presented in a format readable by (norm-aware) agents; 3) a set of protocols (conventions) that agents that are incapable of normative reasoning can use to perform their assigned task; and 4) an active norm enforcement mechanism to see to it that the norms specified for the domain are adhered to and that order and safety is guaranteed in the system. These four elements are combined into a methodology that gives the relations between laws and electronic institutions. Moreover, (formal) methods are specified for the implementation of norm enforcement and the (automatic) creation of protocols (based on constraints specified by the norms).

3.5 Summary of models

In summary, from the above survey, it is clear that a number of these models are complementary, since each focuses its strength on aspects of the organisation the authors deemed more important and some of them are covered in other models, although given different terminologies. For instance, it is clear that the AGR model corresponds to the structural dimension of the MOISE+ model. Also while MOISE+ places emphasis on the structural and functional dimensions of an organisation, it does not consider the interactive dimension. HARMONIA mainly focuses on the normative aspect of an organisation which is also considered in OperA. However, in a more recent model OMNI (organisational model for normative institutions), Dignum et al. (2004) present an integrated framework modelling multi-agent organisations based on the two earlier models OperA and HARMONIA. This model tends to complement the strengths of each of the earlier models, therefore covering relatively well the various dimensions of an organisation. Also comparing the ISLANDER language with AGR and MOISE+, aside from the change in terminology from organisation to institution, ISLANDER focus on institutional aspect of organisations which is further expressed in normative aspects (norms that enforce behaviour) and dialogical aspects (dialogic interactions), while AGR and MOISE+ model structural and functional aspects of an organisation (Coutinho et al., 2005). O-MaSE relied strongly on struc-

	Modelling Dimension			
Model	Structural	Dialogical	Functional	Normative
AGR	++	+	-	-
MOISE+	+++	-	+++	+
O-MaSE	++	-	++	+
HARMONIA	++	+++	+	+++
OperA	++	-	+	+++
ISLANDER	+	+++	-	++
OMNI	++	+++	+	+++

Table 3.1: The relative expressiveness of some organisational Models (adapted from Coutinho et al. (2005))

tural and functional dimensions of an organisation. Table 3.1 summarises these relative strengths and weaknesses.

3.6 Institutional Frameworks

In the previous sections, Section1.3 on page 13 in particular, we have pointed out the focus of this thesis which is the analysis of security requirements that arise from the interaction of various system actors. This is to say that our view is that actions performed by agents in the system could lead to either direct security breaches or create security vulnerabilities. Such actions can vary greatly so that the analysis could be very difficult using the existing approaches we have reviewed. The question here now is that since the early consideration of security in system design is greatly advocated for, how can one possibly analyse the behaviour of agents in the system-to-be at design time? This section presents the institutional framework which this thesis is heavily based upon. This is a formalisation which enables the analysis of actions based on the specified rules of engagement between actors. Based on this formalisation, we shall be expressing the actions and rules that the actors are expected to fulfil in order for us to analyse the consequences of the actions in the light of the security of the system.

This work is in part motivated by the desire to assist the secure system design process, by making it possible *a priori* to determine whether or not a given system specification complies with, or violates the designer's expectations. In order to do this it becomes necessary to define both the means for modelling the system

specifications and a practical reasoning framework through which properties of the specifications may be verified.

We utilise the notion of institutional framework put forward in Cliffe (2007), where an electronic institution serves to describe the rules governing agent behaviour in a multi-agent system. The idea is to provide an explicit, machine processable representation of the norms, rules or regulations which may be used to bring about a set of expected behaviours for agents interacting in a social context. It is assumed here that the norms and their expectations about the behaviour of participating agents are explicit and can be written down in a form which is machine processable. The formalisations components of the institutional framework relevant to this thesis are largely reproduced from Cliffe (2007) where the formalisations were originally presented in order to make this thesis self-contained.

Before we go into details, we would start by defining our usage of the institutional framework terminology.

Definition 2. *In this thesis we use the term* Institutional Framework *to describe a set of rules or regulations and the operational semantics which specify and regulate specific interaction aspects of an organisation. An institutional framework may consist of different types of rules: (a)* constitutive rules *create the possibility for certain activities to take place. They regulate the creation of institutional facts and the modification of the institution. They define how events are interpreted in the institution. (b)* Regulatory rules *regulate antecedent activities by defining what is accepted as correct and incorrect behaviour in terms of obligations, permissions, and prohibitions (Searle, 1995). Institutional frameworks also define the scope of the rules they contain in terms of when (in time), under which conditions, and to which participants and actions they apply (Balke, 2011).*

Principles of Deontic Logic and Normative Positions:

The approach for formalisation presented in this thesis is based on the application of deontic logic. In this section we are presenting the principles behind this logic which we shall be using in the later chapters and sections.

Deontic logic (von Wright, 1951; Hilpinen, 1971) is the branch of symbolic

logic which concerns itself with the investigations of normative concepts, systems of norms, and normative reasoning. Normative concepts include the concepts of obligation (that which we ought to do), permission (that which we are allowed to do), and the forbidden (that which we must not do) and related notions including the concept of right. It has its origin in philosophical logic, applied modal logic, and ethical and legal theory.

Formalisation of legal reasoning in classical logics date back to 1926 (Mally, 1926) with the first formal logic for reasoning with deontic (the study of duties and obligations) properties proposed by Mally (Lokhorst, 2008). This proposed approach has led to an entire field of study as it is been refined and extended in a number of ways yielding various classical logic definitions. One of such definitions is standard deontic logic (SDL), originally accredited to von Wright whose system was first published in von Wright (1951). SDL is a modal logic with operators for permission, obligation, and prohibition. For instance $\mathcal{O}(X)$ expresses the modality obligation or ought which may be read as *it ought to be the case that X is true*, or *it is obligatory that X is true*. This has been used in analysing normative law and normative reasoning in law as illustrated in Jones and Sergot (1996) and Sergot et al. (1986) because it provides a means of analysing and identifying ambiguities in set of legal rules.

3.6.1 Aspects of Institutional Specification

An institution (North, 1990) is modelled as a reactive process which is created, evolves over time through the interpretation of events in some external context (the world), and may be dissolved. Within this process the institution creates states interpretations of effects of changes in the world on the institution, which may in turn affect how the institution changes in the future. We continue by describing how institutional frameworks relate to the real world and how both evolve with time.

Our view of the institutional framework and the real world are presented in Figure 3-1. Both views are represented in terms of system states at various points in time. The real world view on one hand is concerned with the physically observ-

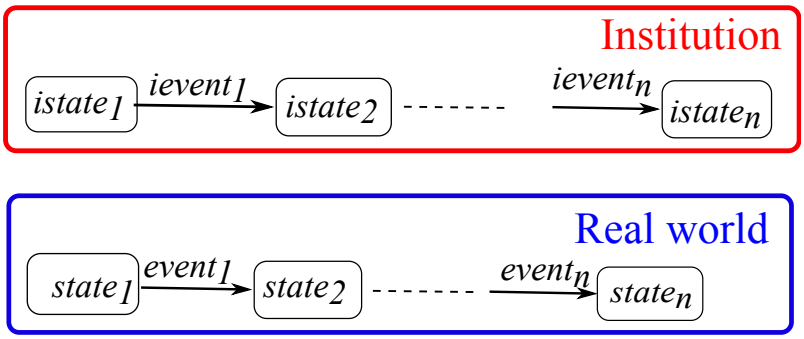


Figure 3-1: Institutional Framework and Real World Transitions

able events that take place in the system. On the other hand, the institutional framework view is concerned only with the events that are relevant in the institution. It is important to note that these two views are not interdependent but rather, they are logically linked. This can best be illustrated by an example: take marriage for an example. Marriage involves a social union or legal contract between two people (real world event). Whether such a union is considered to be marriage or not is a fact that may only be established by the institution of marriage (institutional framework).

Another thing to note from Figure 3-1 is the evolution of the two views over time. Both views start from an initial states which correspond to when the system is initialised. These states subsequently change as a result of events taking place at various points in time. We refer to the states of the institutional framework view as institutional states ($istate_i$) and the real world states as $state_i$. The transitions of these states as described in Figure 3-1 are brought about as events take place either in the institutional framework ($ievent_i$) or in the real world ($event_i$) respectively.

A number of questions arise from the presentation of these views. These include: how the logical link between these two views is specified and the emergence of the institutional framework events, how to determine which real world events are relevant for the institutional framework and which events could be neglected and how to distinguish between the two, and finally, how the transition from states could be described (Balke, 2011).

Most relevant for answering these questions is the work of Searle (Searle, 1995)

in which he distinguished between constitutive and regulatory aspects of an institution. Searle discusses constitutive rules as rules which regulate the creation of institutional facts and the modification of the institution. They define how events are interpreted in the institution. He also introduced the concepts of *brute facts* and *institutional facts* in which brute facts are described as facts that follow from a common-sense understanding of the real world whereas institutional facts are those facts that are only valid within a certain context and depend on human opinions and interpretation. From Figure 3-1, brute facts correspond to real world facts (states), while institutional facts are equivalent to the institutional framework facts (states). These human opinions, according to Searle are shaped by the institutional setting with which the humans interact. This sets the context for the institutional facts and consequently defines whether they are valid and to what extent. In essence, he specifies constitutive rules as rules describing what action executed in one context, *counts-as* performing another action in a second context. One can therefore explain the creation of institutional facts if one takes the physical world as the first context and the institutional framework as the second context and defines when the execution of certain actions or the presence of certain events lead to actions in the second context.

3.6.1.1 Constitutive Aspects of Institutions

In addition to Searle's account of how institutional facts come into being in which he relates the institutional facts and brute facts with the notion of *count as*, Goldman (1976) presents an extensive analysis of actions and events in the real world generating actions and events in a second context (institutional framework in this case). Goldman described four cases where action in one context may be considered to generate actions in another context. These are *causal generation, simple generation, conventional generation and augmentation generation.* Of these four, the conventional generation has been considered the most relevant as it applies to the institutional framework and the *count as* principle from which this thesis draws its inspiration (Cliffe, 2007).

With respect to conventional generation, Figure 3-2 gives an illustration of how this may happen between the real world and the institutional framework. We note that this approach is based on the notion of (i) observable events that capture the

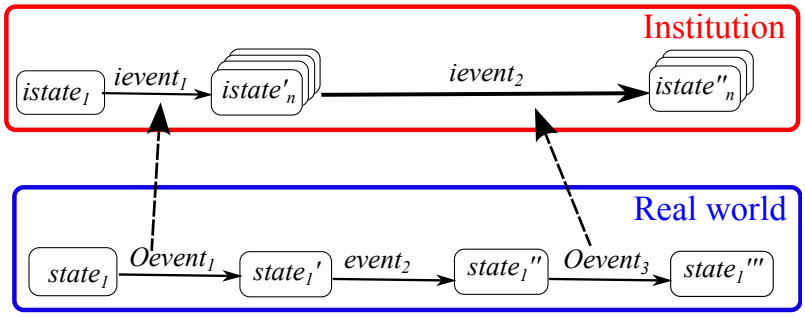


Figure 3-2: Creation of Institutional Events and States through Conventional Generation

notion of real world events and (ii) institutional events that only have meaning within a given context.

Institutional events are not observable, but are created through *conventional generation*, whereby an event in one context *counts as* (Jones and Sergot, 1996) the occurrence of another event in a second context. Taking the real world as the first context and by defining conditions in terms of states, certain real world events may then count as institutional events. Thus, the institutional framework functions as a kind of filter on real world events, selecting only those events that are of relevance for the institutional model and then applying the institutional rules to determine how the institutional state shall evolve over time subject to the occurrence of events. An institutional state is a set of institutional facts that are considered true at some instant. This is illustrated in Figure 3-2 where the $Oevent_i$ stand for observable events, $event_2$ for an unobservable event, and the dotted arrow lines indicate the generation of institutional events. The observable events are events that can be perceived by relevant system participants who would be able to establish the relevance of such an event in the institutional framework. For example, using marriage instance as before, marriage involves the exchange of vows between two people, however, this event has to be observed by say a clergy who has the legal power to establish the institutional fact that these two people are indeed married, and hence change their state of of their social status as married. If the clergy does not observe the event of exchange of vows, and consequently does not bring about the change of state in the institutional framework, the event does not count as marriage and therefore does not have any institutional effect.

Figure 3-2 also illustrates the fact that one observed event in the real world can generate an institutional event that will result in more than one institutional fact (state) in the institutional framework. Taking a football match for example, if one player kicks another player, this can result in both the award of a foul as well as a card. Therefore the observation of a real world event could result in the generation of several states[1] in the institutional framework. Transition from one institutional state to the next state based on the institutional events generated is determined by the constitutive rules. These specify which institutional facts result from which institutional events in which context. They therefore specify the generation, initiation and termination of institutional states.

Event Generation:

The conventional generation of institutional events, where an event within the institution is considered to have been generated as a result of the interpretation of the rules defining the institution, implies two types of events can be generated:

i) those that are generated as a result of the interpretation of events originating from a context external to the institutional framework, such as the environment (in the case of this thesis, the real world organisation where actors interact with information systems and information system security need to be preserved). Within the context of security, we can consider possible origins of such events to include time related events such as time-outs, system logins, and transfer of files from one medium to another

ii) those events that are generated as a result of the interpretation of events within the context of the institution. The internal events could be *institutional events* which cause a transition in the institutional state, or they could be *violation events* that indicate the occurrence of violations.

Violation events could be generated as a result of non fulfilment of an obligation or out-rightly from the occurrence of non-permitted event.

[1]In this thesis, the terms *states* and *facts* are used synonymously with respect to the descriptions of the institutional framework and conventional generation.

3.6.1.2 Regulative Aspects of Institutions

In the preceding section, we presented the different aspects of constitutive rules which are associated with the semantics of the institution. With this, participants in an institutional framework can observe sequence of events which have occurred in a context external to the institution and then make interpretations based on the institutional events which should be considered to have occurred. However, constitutive rules alone cannot allow for the determination of which events are considered to be bad or good within the institutional framework. Hence there is a need for a framework which will account for what is perceived as institutionally correct and incorrect behaviour. Regulative aspects of the institution which make this possible consist of obligations, permissions and prohibitions.

Permission and Prohibition

Permissions, as regulatory aspect of institution, indicate that some actions and state of affairs within an institution are considered acceptable or desirable. Prohibitions, on the other hand, can be seen as the dual or counterpart indicating actions which are not permitted. Hence if an action is not permitted, it is considered prohibited or forbidden. This view of expressing one in terms of the other is taken in both legal theory and formal logic (Vranes, 2006; Prisacariu and Schneider, 2007). We therefore observe this view in this thesis as we explicitly represent only permissions and the absence of permission is defined implicitly to be prohibition. With this, we can easily separate which actions and situations are good or bad since an action that is not permitted is simply prohibited.

Now that a way has been determined in which a distinction can be made between good and bad actions and situations in institution specifications, the question that need to be addressed is what subject(s) does this operator (permission) apply to? Two foci are presented in Cliffe (2007) for the application of permission:

i. permissions on the state of the institution which may take the form "it is [not] permitted for state s to come about", and

ii. permission on institution actions and events which may take the form "it is

[not] permitted for action α to be performed"

Following these, the focus in this thesis is going to be on actions and events. The reason for this is that we would like to be able to reason not only about the presence or absence of violations, but importantly about how the violations came about. This is important for us in this thesis as we typically wish to consider the occurrence of certain states of violations and examine the actions that led to such states of violation on one hand and the consequences of such states on the system on the other hand. The implication of choosing a single operator from the view point of reasoning about models is that a *closed world* is assumed. That is in this case, it is assumed that all cases in which an action is permitted must be known in order to infer that an action is prohibited by the absence of permission. This is sufficient for us for the purpose of analysis and verification of our institutional models since we are dealing with complete models.

Obligations and Violations

In the previous section, we established permissions and prohibitions as means of asserting about which events are allowed or not allowed at a specific point in time. However, we are also interested in being able to express the situation where an action/event is required to be triggered at certain points in time. This is achieved through the notion of obligations which can be expressed as *immediate obligation*, *prioritised obligation*, and *obligation with deadlines* (Dignum et al., 1996).

Which ever way obligations may be expressed, it is important as with permissions and prohibitions to identify the subject(s) of obligations. This may be either institutional states that a participant ought or ought not to bring about at a given time, or actions that ought to be performed by a participant before another action or deadline takes place. Reasoning in the same way as in the case of permissions and obligations, the choice here is for an event/action approach for association of obligations.

The choice of event/action as the focus for obligations brings to the front the question of how obligations may be associated with events. Two possibilities come to mind: i) Association of obligations with a particular actor in the system

which may take the form "actor A ought to do action α". This is seen as a directed obligation since a particular actor is expected to perform a certain action. This looks intuitive and the right kind of obligation association but it is not convenient to express when a group of actors are involved. This will mean expressing the obligation in terms of each actor in the group which will be cumbersome and problematic from modelling perspective. ii) Association of obligations with particular action[2] which may take the form "it ought to be the case that action α is performed". In this case obligation is associated with an action which in turn will be associated to an actor as may be defined by the constitutive rule of the institution. This undirected approach solves the problem identified in the directed approach. From these, we focus on the association of obligations with events which in turn may be performed by any actor.

Closely related to the notions of permission/prohibition and obligation is the notion of violation. This is a case where an actor performs an action that is not permitted or fail to fulfil the required obligation. This usually attract some form of penalty such as sanctions as a means of deterrence and compliance. Compliance may be enforced either privately, that is indirectly as a result of changes in actors' behaviour, or publicly, that is directly by the application of a sanction within the context of the institution. In the case of public enforcement, the imposition of the sanction may either be optional in which case the violation only opens the possibility for the sanctioning actor to bring about the sanction, or mandatory in which case it is obligatory for the sanctioning agent bring about the sanction. Both of these cases may be expressed as normative rules (either a permission or an obligation to perform the sanction), which simply follow as consequences. This way sanctions can be naturally expressed in the language of the norms which they enforce.

Institutional Power and Permission

The notions of power and permission can be understood from Searle's notion of *counts as* and the corresponding notion of *conventional generation* by Goldman which formally account for the relationship between *brute fact* and *institutional*

[2]The terms *event* and *action* are used interchangeably in this thesis.

fact as discussed in sub Section 3.6.1.1. In formalising these notions, Jones and Sergot (1996) identified the variations in which institutionalised power can be seen and argued for the necessity of distinguishing between. These are:

i) legal power (also referred to as normative or institutional power in the institutional framework context)
ii) physical power - the physical capability to carry out the acts in the physical environment, and
iii) the permission to carry out those acts.

Applying these concepts to the principles of *conventional generation* and *counts-as*, it can be established that legal power specifies who has the legal capability to bring about action in the institutional context, therefore acting as a constraint on conventional generation of actions. For instance, in the real world we may be concerned with our capability to directly bring about some state of affairs physically, however in the legal world the concern is about when the creation of certain states is considered valid. This is also the concern in the context of institutional framework. While, the notion of power is dependent on the capability of an agent to carry out an action and the interpretation of such an action, permission is independent of capability. An agent may be empowered to perform an action but may not be permitted to perform the action at certain points in time. For example a user may be empowered to login to a system but may not be permitted to do that outside the official working hours.

Jones and Sergot (1996) formalised this notion of institutionalised power as a means for modelling conventional generation using the logic of consequence of action expressed as $E_x A \Rightarrow_s E_y F$, where $\Rightarrow_s$ designates the notion of consequence (*count-as*) with respect to the institution s. This expression accounts for the notion of agency in the sense that it allows for the association of an agent (x) with a real world action E_x which brings about a change in the state of affairs in the real world to a state A. From the expression, it also means that within the institution s, this action might also bring about a change in state of the institution to state F. The agent y can be the institution itself or can be same as agent x. For example, using the the case of marriage, the priest may play the role of x who physically performs the marriage ceremony, but the normative/institutional

relation of being married created by the church establishes the marriage as a fact.

3.6.1.3 Institutional Framework Initiation and Termination

An institution may only be considered to be in force only for a given time interval. This is a temporal scope marked by the initiation and termination of the institution. Within the specification of an institution, it is important to define when the institution has an effect and when it does not. This can be done by specifying when, how and possibly with which parameters an institution is created and when and how it is destroyed.

Creation of an institution is defined as a transition from a point in time when no rules defined within the institution have any effect, to a subsequent point in time when those rules have an effect. Also, the destruction an institution is the transition from a point at which the institutional rules may have an effect to a point at which they may no longer have any effect. The institution is therefore considered to be in force between those two points in time.

3.6.1.4 Time

Following from the last subsection, an important aspect that must be considered when designing a system that models action and change such as the institutional framework is the handling of the problem of time representation. A number of ways have been found useful as pointed out in Allen (1991) which include methods based on explicit dating, intervals, and temporal logics. In this thesis, since the focus is on events and states of the institution, no assumption is made about the real time interval between events, but rather, time is treated as a set of ordered instances denoting the points in which real world events occur. The ability to express and model the notion of time in this way allows for a chronological analysis of event and chains of events. This is particularly relevant as we apply the framework to the security domain in which the interval between events is considered important (Beres et al., 2009).

3.6.2 Formal Definition of Institutional Framework

This section presents the formal definition of the components of the institutional framework relevant to this thesis. The full formalisations are presented in Cliffe (2007).

Definition 3. *An institutional framework is defined as a 5-tuple* $\mathcal{I} := \langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{G}, \Delta \rangle$ *where* $\mathcal{E}$ *is a set of events,* $\mathcal{F}$ *a set of fluents,* $\mathcal{C}$ *a set of causal rules,* $\mathcal{G}$ *a set of generation rules and an initial state* Δ*.*

The institution defines the interplay between these components as the institution evolves over time due to the interaction of its participants. We further define each of these components as follows:

Events $\mathcal{E}$:

A set of event symbols $e \in \mathcal{E}$ are defined in the framework, each denoting a type of action that may occur which may fall into either of the disjoint subsets: $\mathcal{E}_{ex}$ consisting of exogenous events and $\mathcal{E}_{inst}$ consisting of institutional events with $\mathcal{E} = \mathcal{E}_{inst} \cup \mathcal{E}_{ex}$ and $\mathcal{E}_{inst} \cap \mathcal{E}_{ex} = \emptyset$.

Exogenous events $\mathcal{E}_{ex}$:

These are events that take place outside the scope or context of the institution but whose occurrence trigger institutional events in accordance with the counts-as principle discussed earlier. These may include actors communicating with actors such as *reveal(x,y,password)*, actors interaction with a system such as *login(x,system)*, and other externally defined events such as time-outs. Another term used to refer to exogenous events is *observed events* due to the fact that any event that would have an effect in the institution must be observable and considered relevant in the institution. This thesis uses the terms *exogenous* events and *observed* events synonymously.

Additionally, exogenous events also consist of a set of creation events $\mathcal{E}_{+} \subseteq \mathcal{E}_{ex}$ which account for the creation of an institution.

Institutional events $\mathcal{E}_{inst}$:

These contains the events that are generated in the institutional framework as a result of occurrence of exogenous events. $\mathcal{E}_{inst}$ is further classified into institutional actions $\mathcal{E}_{act} \subseteq \mathcal{E}_{inst}$ that capture changes in institutional state, and violation events $\mathcal{E}_{viol} \subseteq \mathcal{E}_{inst}$, that signal the occurrence of violations which could come about as a result of explicit occurrence of prohibited events or from the failure to fulfil obligations as discussed earlier. In the framework, the set of violation events is defined such that there exists at least one violation event corresponding to each institutional action and each exogenous event as follows: $\forall e \in \mathcal{E}_{act} \cup \mathcal{E}_{ex}\, \exists viol(e) \in \mathcal{E}_{viol}$. Therefore institutional events $\mathcal{E}_{inst} = \mathcal{E}_{act} \cup \mathcal{E}_{viol}$ and $\mathcal{E}_{act} \cap \mathcal{E}_{viol} = \emptyset$.

As with exogenous events, institutional events also consist of a set of dissolution events $\mathcal{E}_{-} \subseteq \mathcal{E}_{act}$ which account for the termination of an institution.

Fluents $\mathcal{F}$:

A fluent is a property of the institution that holds after it is initiated and ceases to hold when terminated. Two types of fluents are distinguished for the institutional framework: *institutional fluents* which denote normative properties of the institutional state such as permissions, powers and obligations, and *domain fluents* which correspond to properties which are specific to the institutional framework itself. In both cases, fluents are modelled as propositions which may be true or false in a given institutional state. The set of normative fluents is broken down into sets of fluents for powers $\mathcal{W}$, permissions $\mathcal{P}$ and obligations $\mathcal{O}$ as follows:

- Power $\mathcal{W}$: Consists of institutional power fluents of the form $pow(e) : e \in \mathcal{E}_{act}$. This denotes the capability of some action e to be brought about in the institutional framework.

- Permission $\mathcal{P}$: This is composed of a set of event permission fluents of the form $perm(e) : e \in \mathcal{E}_{act} \cup \mathcal{E}_{ex}$. This permits the action e to be brought about. A *forbidden* event is simply treated as the absence of permission for that event to be brought about.

- Obligation $\mathcal{O}$: Obligations fluents of the form $obl(e, d, v)$ with $e \in \mathcal{E}, d \in$

$\mathcal{E}, v \in \mathcal{E}_{viol}$ denotes that event e should occur before event d else violation v will be generated.

Domain fluents $\mathcal{D}$ is defined to include all properties of the institutional state that are specific to the institution which may be true.

The set of all institutional fluents $\mathcal{F}$ which may hold true in a given institutional state is defined as follows: $\mathcal{F} = \mathcal{W} \cup \mathcal{P} \cup \mathcal{O} \cup \mathcal{D}$ where $\mathcal{W} \cap \mathcal{P} \cap \mathcal{O} \cap \mathcal{D} = \emptyset$. These are the fluents which may be created within the context of the institution. However, considering the life-cycle of the institution, it is necessary to define another fluent *live* which accounts for the creation of the institution. This fluent is treated differently from the institutional fluents described earlier in the sense that its definition is not within the semantics of the institution itself. Therefore the set $\mathcal{F}^*$ of all institutional fluents including the external fluents is defined as $\mathcal{F}^* = \mathcal{F} \cup \{live\}$.

States

Having defined the set of all institutional fluents, it is now possible to define the possible states of the institution that could hold as $\Sigma = 2^{\mathcal{F}^*}$. However, due to the effect that institutional rules may have on multiple institutional states, not all of these states may be reachable. This effect of the institutional rules over a specific set of states is hereby accounted for by defining the set of state formulae $\mathcal{X}$ over a given set of institutional fluent properties $\mathcal{F}$ which may be true while the institution is active as $\mathcal{X} = 2^{\mathcal{F} \cup \neg\mathcal{F}}$. $\mathcal{F} \cup \neg\mathcal{F}$ specifies all the set of literals including negations over $\mathcal{F}$.

Consequence and Generation Rules

Here, we present the formalisation of the consequence relation which describes the initiation and termination of fluents as a result of the performance of a given action in a state matching some expression. This is defined by a function which is expressed as $\mathcal{C} \colon \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{F}} \times 2^{\mathcal{F}}$. Given an event $e \in \mathcal{E}$, the first set in the range of the function represents the fluents that are initiated by the event. This

is denoted by $\mathcal{C}^{\uparrow}(\phi, e)$ for the event e in a state matching ϕ. The second set in the range of the function describes the fluents that are terminated by the event and denoted $\mathcal{C}^{\downarrow}(\phi, e)$.

Similarly, an event generation function $\mathcal{G}\colon \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{E}_{inst}}$ is also defined by each institution to describe the count-as notion explained earlier. This describes when the occurrence of one event generates another.

3.7 Summary of Chapter

In this chapter we presented the approaches for modelling organisations with institutions. We introduced the concept of institutions and discussed how it is related with organisation. Institutions shape organisations by providing a normative context within which organisations seek their ends. Organisations on the other hand reproduce institutions by conforming to their norms and from time to time seek to change institutions. Institutions provide a normative context both limiting and facilitating social change (Bouma, 1998). We then reviewed the existing approaches for modelling institutions and summarised them in terms of their strengths and weaknesses.

We introduced the institutional framework upon which our work is based. The institutional framework is inspired by deontic logic, that is the logic of power, permissions and obligations. In Chapter 1 we established the connection between security goals of confidentiality, integrity and availability (especially the first two) with the notions of permissions (and prohibitions) and obligations. Since the institutional framework is inspired by these concepts it becomes the choice approach for this work. The institutional framework consist of two layers of abstraction: the institutional layer and the real world layer. Based on the count-as principle, institutional events are generated by the observed events in the real world layer resulting in the change of the institutional state. We discussed the various aspects of the institutional specification relevant to our work and presented the formal definition of the institutional framework.

In the next chapter we present the realisation of the institutional framework as a computational model which will provide us with the information needed for modelling and analysing our security requirements scenarios.

Chapter 4

Computational Logic Reasoning Approaches

4.1 Introduction

The goal of this thesis is the presentation of a computational approach to the verification of security requirements and properties. Since we are approaching the problem of information security primarily from the human factor point of view, the previous chapters have presented the notions of institutions and organisations which are both concerned with the structures that guide human interaction. We have also presented the institutional framework which allows us to monitor behaviours for "correct" and "incorrect" behaviours. This chapter presents the computational logic approaches that would make it possible to reason about our security requirements thereby enabling us to perform verification tasks on elicited security requirements computationally.

4.2 Answer Set Programming

Answer set programming (ASP) (Gelfond and Lifschitz, 1988; Marek and Truszczyński, 1991; Baral, 2003) is a declarative logic programming paradigm that allows reasoning about possible real world views in the absence of complete information. ASP allows a programmer to specify "what" needs to be done without specifying "how" it needs to be achieved. Thanks to its formal semantics called the answer set semantics (Baral, 2003), and the existence of efficient heuristic solvers (such as *smodels* (Niemelä and Simons, 1997), *dlv* (Eiter et al., 1999; Leone et al., 2002) and *clingo* (Gebser et al., 2008)), answer set programming provides an

excellent basis from which derived models may be queried, hence an approach for solving search problems (Lifschitz, 2008; East and Truszczyński, 2006). To solve a given search problem in ASP, one designs a logic theory whose *models* represent solutions to the problem. Therefore the primary computational task is the problem of finding models (*answer sets* in ASP terminology) rather than proofs.

One of the advantages of using ASP is that its rigorous and formal semantics allow for reasoning in formal ways about the semantics of programs. An answer set program can be seen as a formalisation of the underlying reasoning problem in its own right, with the advantage of being able to execute this formalisation directly through the use of answer set solvers.

ASP is particularly suited to model-based reasoning by allowing domain and problem-specific knowledge, including incomplete knowledge, defaults, and preferences to be represented in an intuitive and natural way (Brewka et al., 2011). It is an intuitive non-monotonic programming language suitable for modelling, reasoning and verification tasks. An answer set programming problem typically starts with the definition of a domain or class of problem about which we wish to reason. Such definitions (written as answer-set programs) are constructed in such a way that each possible model in the domain corresponds to an answer set of the program. Queries may then be constructed over this domain in order to determine if particular models are valid according to the definition by extending the program to limit the answer sets produced to those which match the models we are interested in.

Processing answer set programs involves the use of tools which support several basic reasoning tasks ranging from computing a single answer set, determination of non existence of answer sets, to computing all possible answer sets. Most tools also support the task of *cautious reasoning* which has to do with deciding whether an atom is true in every answer set.

ASP is typically processed in two stages as shown in Figure 4-1; first is the replacement of the predicate program with an equivalent propositional program through the process of variable replacement, also known as *grounding*. Grounding is a process of translating a program containing variables into a program

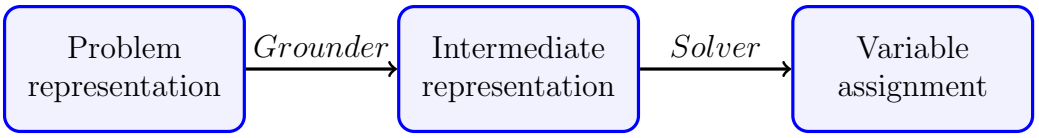


Figure 4-1: ASP System Architecture (Gebser et al., 2008)

containing no variables. The ground program is then solved by a propositional ASP solver which gives answer sets for the program. We use a ASP solver called *clingo* in this thesis. Clingo is a combination of the *clasp* solver and *gringo* which is a state of the art grounder that has a rich set of features that make writing logic programs more efficient (Gebser et al., 2008).

4.2.1 How ASP compares with similar formalisms

There are many other formalisms for solving search problems using logic reasoning procedures beside ASP. In this subsection, we discuss how ASP compares with some of the related and relevant formalisms for declarative problem solving.

4.2.1.1 ASP and SATISFIABILITY (SAT) solving:

Both SAT solving (Eén and Sörensson, 2004) and core ASP apply in principle to the same problems, however they are different in a number of ways. First is how problem specifications and data are handled. ASP because of its support for variables that range over finite domains, enables uniform and compact specification of problems independently of data. Grounded instances of the problem specifications are produced for ASP solvers based on the problem specification and the data input. This separation of problem specifications from data has several advantages including facilitation of debugging and testing, support for optimization and development of reusable problem modules. In SAT however, such separation of problem specification and data is not available, rather the two are integrated into programs that generate satisfiability instances to be solved. This makes development of software engineering techniques for SAT more difficult. Secondly, although any problem that can be modelled in SAT can be modelled equally well in ASP, problems such as reachability in graphs are easier to cast in ASP than in SAT solving since their representation for SAT solving results in larger instances that slow down solving. Also the ASP language offers support through

ASP solvers for constructs such as *minimized* disjunction, aggregates and priorities that are useful in practical applications. In SAT solving, specialised ad-hoc treatments are required for these constructs and concise representations are not even possible for some of them (Brewka et al., 2011).

4.2.1.2 ASP and Prolog:

Although Prolog is the most widely known logic programming language it has a number of limitations both in concept and design which makes it unsuitable for many knowledge representation real world reasoning tasks. ASP and Prolog are similar in syntax with semantic connections too, for instance, if Prolog returns "yes" (or "no") to a ground query for a program Π, then the query belongs (or does not belong) to the answer set of the program. In spite of these connections, ASP and Prolog are actually quite different when compared in terms of suitability and ease of expression in a problem domain. It is worth noting also that Prolog was actually designed as a general purpose, Turing-complete programming language. It is more procedural in nature than declarative, using function symbols for nested terms to build potentially infinite data structures. Prolog solutions are computed by query answering and thus proof finding. In contrast, ASP was not conceived for such generality but works over a finite domain of data and solutions are encoded in models (that is answer sets). Therefore where Prolog is concerned with proof finding, ASP is concerned with model finding.

Logic programming languages, Prolog inclusive, have to deal with the problem of negation in some ways. Existing notions of negation are *negation as failure (NAF)*, that is **not** p is true if p cannot be proved, and *classical negation* $\neg p$ that is every proposition can either be true or false and not both. With these, the most common approach to handling negation is to compute negation as failure, that is **not** p is true if p can be proved using the current program; and to characterise this as classical negation. When using Prolog to model real world reasoning, it is essentially assumed that every knowledge about the world is contained in the program. This creates a problem referred to as the closed world assumption. By characterising NAF as classical negation anything that cannot be proven to be true is known to be false.

ASP semantics naturally allows negation to be treated in two different forms: *negation as failure* and *constraint-based negation*. Negation as failure, (i.e. p cannot be proven to be true) is characterised as epistemic negation, (i.e. p is not known to be true). Constraint-based negation on the other hand prevents certain combinations of atoms from being simultaneously true in any answer set by introducing constraints. This is characterised as classical negation as it is possible to prevent p and $\neg p$ from being simultaneously true. This makes reasoning about incomplete information possible, and is supported by the intuition that "I do not know that P is true" (auto-epistemic negation) and "I know that P is not true" (classical negation) are fundamentally different.

Another area where ASP is different is in reasoning about multiple world views. The semantics of ASP naturally give rise to multiple possible world views in which the program is consistent thereby making it possible reason in terms of the world views as expressed in the program. This type of reasoning is only indirectly possible in Prolog using the cut operator, however due to the procedural nature of Prolog rules, this can lead to confusion as the multiple possible views may manifest themselves differently depending on the query asked. For instance, in ASP terms Prolog would answer a query on a as true if there is at least one answer set in which a is true. However because there is no way of knowing in which answer set this is true, a subsequent query on b might also return true. It would not be possible to infer if a and b could be simultaneously true without another query.

Finally, similar to the point regarding SAT solving, there is a strong link between the Prolog language used for querying a knowledge base and the language in which the knowledge base is represented. Knowledge is essentially stored in a form that for example, to answer a question about c we must answer questions about b and c, notated as $c :- a, b.$ In contrast, ASP strongly distinguishes between the program and the query language. The program itself generates answer sets which the query language then constrains to result only in those that are consistent with the query. This separation helps clarify the conceptual difference between representing or manipulating data and querying it. It also allows the query semantics to make full use of the multiple possible world views provided (Cliffe, 2007).

4.2.2 *AnsProlog*

AnsProlog, a short form for "*Pro*gramming in *log*ic with *Ans*wer sets" (Baral, 2003, p. 3), is a declarative programming language mainly used for knowledge representation which allows for an intuitive rather than algorithmic description of a problem and the requirements to be fulfilled by the solution to the problem. The corresponding solutions to the problem are called the *answer sets* of the *AnsProlog* program describing the the problem. These are defined through (a variant or extension of) the stable model semantics (Gelfond and Lifschitz, 1988) and computed using a set of tools referred to as answer set solvers. Available solvers include clingo (Gebser et al., 2008), SMODELS first introduced by Niemelä and Simons (Niemelä and Simons, 1997), the DVL system (Leone et al., 2002), ASSAT (Lin and Zhao, 2004), and CModels (Lierler and Maratea, 2004; Lierler, 2005). *AnsProlog* is non-monotonic, that is conclusions drawn on the basis of given knowledge can be retracted when new knowledge is available. This is because the set of conclusions reached on the basis of a given knowledge base, does not necessarily increase (rather, it can shrink) with the size of the knowledge base itself. This contrasts with the standard monotonic logic frameworks such as classical first-order logic, whose inferences, being deductively valid, remains valid in the presence of new information (Antonelli, 2012).

AnsProlog has the advantage that it allows for the specification of both the problem as well as the query for the answer set as an executable program resulting in a more straightforward verification and validation task. This contrasts with related approaches such as event calculus and C^+ which create a gap between specification and verification language. The next subsection presents the syntax of *AnsProlog*.

4.2.2.1 ASP Syntax

In this section we present the syntax of *AnsProlog* - the underlying logical language for ASP if interpreted using the answer set semantics (Gelfond and Lifschitz, 1988). There are a number of syntactic variants of ASP in existence among which is a broad notion known as $AnsProlog^*$ as described in Baral (2003). Out of the many subgroups of $AnsProlog^*$, this thesis is limited only to $AnsProlog^{\perp}$

and we will describe its syntax and functions in this section.

The language of an ASP program (*AnsProlog*) consists of;

- a set *variables*: sequence of letters starting with an upper case letter$X, Y, Z, ...$
- a set of *constants*: sequence of characters starting with a lower-case letter $a, b, c, ...$
- a set of n-ary function symbols: sequence of characters starting with a lower-case letter $f, g, h, ..$ followed by a bracketed list of one or more arguments such as terms.
- a set of n-ary predicate symbols: sequence of characters (the predicate name) starting with a lower-case letter $p, q, ...$ followed by a bracketed list of zero or more arguments. Brackets are omitted where there are zero arguments.

The above alphabet forms the basis for an ASP program. This alphabet is used to the define terms, atoms, literals, and rules which make up a program as described below:

- A *term*: This can be a variable, a constant, or a function $\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ where f is an n-ary function symbol and $t_i s$ are terms

- An *atom*: This takes the form $\mathtt{p_n}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ such that $t_i s$ are terms and p_n is a predicate with n arguments. If n is zero, $\mathtt{p_0}$ is also an atom. This can be assigned a truth value of either *true* or $false$ and can also appear in negated forms.

- A *literal*: in $AnsProlog^*$, this can be an atom $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ or its classical negation $\neg\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ (meaning that $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ can be shown not to be true). Negated literals are omitted in $AnsProlog^{\perp}$, hence all literals we shall be using in this thesis are positive literals.

- A $naf - literal$: This is an extended literal in $AnsProlog^{\perp}$ which is expressed as the literal $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ or its negation-as-failure **not** $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ which means that $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ cannot be proven to be true. Negation-as-failure denoted **not** is therefore different from classical negation denoted $\neg$ where falsity needs to be proven.

- *ground* terms (atoms, literals) : These are terms (atoms, literals respectively) without variables.
- *rule*: $AnsProlog^{\perp}$ program is made up of rules of the form

  L.
  or
  $H \leftarrow L_0, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$.
  where H is a literal or $\perp$ and L, L_i's, $1 \leq i \leq n$ are negation-as-failure literals .

  A rule r consist of a *head* and a *body*. The body of a rule r can be divided into the set of non negated atoms $B^+(r)$ and the set of atoms that appear negated $B^-(r)$. From the interpretation of negation-as-failure, a rule in an $AnsProlog$ program can be read the following way: "if we know $B^+(r)$ and we do not know $B^-(r)$, then the head of the rule can be assumed."

  From the description of a rule above, H is the head of the rule denoted by $Head(r)$ and $\{L_0, \ldots, L_n\}$ is the body of the rule denoted by $Body(r)$. If the head of the rule is empty, such a rule is referred to as a *constraint*. The head is either expressed using $\perp$ or simply eliminated. The rule can simply be written as: $\leftarrow L_0, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$. For the $AnsProlog$ program to be true, the constraints of this program need to be not true.

  On another hand, a rule can have an empty body. Such a rule is called a *fact*. This is written without the "$\leftarrow$" sign as: L. The truth value of a fact must be true.

  If all literals in the body and head of the rule are ground, the rule is said to be grounded.

Definition 4. *For a program* Π*, the set of all ground terms which can be formed with constants and function symbols in* Π *is called the Herbrand Universe of* Π *denoted as* $\mathcal{U}_\Pi$.

For example: Consider a program Π consisting of variables X, Y; constants a, b;

function symbol f of arity 1; and predicate symbol p of arity 1:

$$\mathcal{U}_\Pi = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}.$$

Definition 5. *The Herbrand Base of a program* Π *denoted* $\mathcal{B}_\Pi$ *is the set of all ground atoms which can be formed by applying elements of* $\mathcal{U}_\Pi$ *to the arguments of the predicate symbols in* Π*.*

$$\mathcal{B}_\Pi = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))), \\ p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}.$$

Definition 6. *A program* Π *is said to be grounded when each possible ground term in* $\mathcal{U}_\Pi$ *is applied to each variable in each rule of* Π*.*

For example, given the following unground program:

$$\Pi = \begin{cases} login(bob). \\ login(alice). \\ has\_access(X) \leftarrow login(X). \end{cases} \tag{6.1}$$

The Herbrand Universe $\mathcal{U}_\Pi$ of program Π includes the terms *bob* and *alice* which are used to expand the variable X in the third rule. The Herbrand Base $\mathcal{B}_\Pi$ of program Π can also be derived to include the atoms $login(bob)$, $login(alice)$, $has\_access(bob)$ and $has\_access(alice)$.

With the $\mathcal{U}_\Pi$ and $\mathcal{B}_\Pi$ computed, replacing each variable symbol by one of the terms in $\mathcal{U}_\Pi$ gives a set of ground instances of a rule. In the case where an atom contains multiple variables then each permutation of the possible values for each variable is included. The ground version of a program Π, written as $ground(\Pi)$, is the set of all ground instances of all the rules in the original program. Example

6.1 above would yield the following ground program:

$$ground(\Pi) = \begin{cases} login(bob). \\ login(alice). \\ has\_access(bob) \leftarrow login(bob). \\ has\_access(alice) \leftarrow login(alice). \end{cases}$$

As illustrated in Definitions 4 and 5, an unground program containing function symbols will result in infinite Herbrand Universe and Herbrand Base. This will lead to an infinite number of grounded rules, a case which is undesirable. Current ASP solvers operate only on ground programs with finite sets of rules we therefore constrain our programs to those with finite set of rules. To do this, two constraints - *range restriction* and *domain restriction* properties - are introduced on the structure of rules and programs. These are defined as follows:

Definition 7. *An unground rule is range restricted, if each variable in the rule appears in at least one positive atom (not negated by negation as failure) in the body of the rule. A program* Π *is range restricted if all of its rules are range restricted.*

The range restriction property requires each variable to be associated with one or more predicates in the body of the rule.

The second property, *domain restriction*, applies to the whole program and defined as follows;

Definition 8.

i) A rule is domain restricted if every variable which appears in the rule also appears in a positive domain predicate in the body of the rule.

ii) A program Π *is domain restricted if all rules of the program are domain restricted.*

iii) A predicate is a domain predicate if a ground atom derived from that predicate appears in the head of at least one rule with an empty body (as a fact) and the predicate does not appear in the head of any rules with non-empty bodies.

For example, the program:

```
q(a).
q(f(X))  :-  q(X).
```

is range restricted. However, it is not domain restricted because the predicate `q` appears in the head of both a domain restricted rule and a non-domain restricted rule.

Having presented some of the syntactic issues of ASP relevant to us, we note that while variables allow a great deal of flexibility and syntactic clarity of programs, grounding process may generate a number of rules that may be exponentially larger than the original program. Since the goal of an *AnsProlog* program is to find the answer sets to the program, we would now turn to look at the semantics of ASP which would make this possible.

4.2.2.2 Semantics of $AnsProlog^{\perp}$ Programs

As noted earlier, an answer set program consists of a set of statements, called rules where each rule $H \leftarrow B$ consists of two parts: a head literal H and the body B made up of a set of literals. Intuitively, this means that: "if all the elements of B are true, so is the head H" or "H is supported if all elements of B are considered to be true". This form of reasoning is referred to as the *minimal model semantics*.

In order to define *models* of a program, the Herbrand interpretation of an $AnsProlog^{\perp}$ program Π is defined as any subset $I \subseteq \mathcal{B}_{\Pi}$ of its Herbrand base. Models of the program Π are defined as follows:

Definition 9. *Given a ground program consisting of rules of the form: $h \leftarrow B \in \Pi$ (which defines a constraint when h is $\perp$), where B is the set of (non-negated) literals in the body of the rule and h is a literal (if not $\perp$); A Herbrand interpretation $I \subset \mathcal{B}_{\Pi}$ of the program Π is a model of the program Π iff for each*

rule of the program the following is true:

$$l \leftarrow B \in \Pi \begin{cases} if\ l \equiv \bot\ and\ B \nsubseteq I \\ if\ l \not\equiv \bot\ and\ B \subseteq I \implies h \in I \end{cases}$$

A model M is a minimal *model of Π if, given the set of all models of Π:* $\{M_1, \ldots, M_n\}, \nexists j, 1 \leq j \leq n \mid M_j \subset M$.

Definition 9 does not take negation-as-failure into account. This means that the rules of the program Π may contain negated literals which must not be in the interpretation of the program for the rules to be supported. A reduct or transformation is therefore necessary in order to find the minimal models for programs containing negation-as-failure. This transformation is referred to as the *Gelfond-Lifschitz transformation* (Gelfond and Lifschitz, 1988) and defined as follows:

Definition 10. *Let Π be a ground $AnsProlog^{\bot}$ program. The Gelfond-Lifschitz transformation of Π with respect to an interpretation I where $I \subseteq \mathcal{B}_\Pi$, is the program Π^I containing rules $l \leftarrow B$ such that for all rules of the form $l \leftarrow B, \textbf{not}\ C \in \Pi, C \cap I = \emptyset$ and B and C being sets of literals.*

The reduced program includes all rules of the original program which do not contain negated literals in the interpretation.

With the concepts of *minimal model* and *reduced program* defined the answer set of the program is defined as:

Definition 11. *Let Π be a ground $AnsProlog^{\bot}$ program. A set of ground atoms $I \subseteq \mathcal{B}_\Pi$ is an answer set of Π iff I is a minimal model of Π^I.*

The strength of answer set programming lies in the nondeterministic nature of negation-as-failure, in which a program could result in several answer sets which are all acceptable solutions to the problem that has been modelled. The set of answer sets for a program Π is referred to as $\mathcal{A}_\Pi$.

The concepts introduced and explained here are better illustrated with an example:

Given the following program:

$$\Pi = \begin{cases} \mathtt{x} \leftarrow . \\ \mathtt{y} \leftarrow \mathtt{x}, \textbf{not } \mathtt{z}. \\ \mathtt{z} \leftarrow \mathtt{x}, \textbf{not } \mathtt{y}. \end{cases}$$

The Herbrand base $\mathcal{B}_\Pi$ of this program which is the set of all atoms used in rules in Π consists of the atoms in $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.

The Herbrand interpretation of the program Π yields the following:

$$\{\{\}, \{\mathtt{x}\}, \{\mathtt{y}\}, \{\mathtt{z}\}, \{\mathtt{x}, \mathtt{y}\}, \{\mathtt{x}, \mathtt{z}\}, \{\mathtt{y}, \mathtt{z}\}, \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}\}$$

We can now use these computed information to determine the models of the program as follows:

Given the interpretation $R = \{\mathtt{x}, \mathtt{z}\}$, and the Gelfond Lifschitz transformation:

$$\Pi^R = \begin{cases} \mathtt{x} \leftarrow . \\ \mathtt{z} \leftarrow \mathtt{x}. \end{cases}$$

The interpretation R is a model of Π, since the atoms in R are supported by both rules. This interpretation is also a minimal model with respect to Π^R as it includes only atoms supported by the program.

In contrast, consider the interpretation $Q = \{\mathtt{y}\}$ and the transformation:

$$\Pi^Q = \begin{cases} \mathtt{x} \leftarrow . \\ \mathtt{y} \leftarrow \mathtt{x}. \end{cases}$$

Here, the transformation Q does not include the atom $\mathtt{x}$ which is supported by Π^Q, therefore the transformation Q is not a model of Π^Q.

Lastly, an interpretation could be a model but not necessarily a minimal model of the transformation. This example illustrates this:

Consider the interpretation $S = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and the transformation Π^S:

$$\mathtt{x} \leftarrow .$$

Here, since the interpretation S includes $\mathtt{x}$, it is a model of Π^S but it is not a minimal model because it includes the atoms $\{\mathtt{y}, \mathtt{z}\}$ which are not supported by the transformation Π^S. This is also true for the empty set $\{\}$.

Therefore for the above program Π, the set of its answer sets $\mathcal{A}_\Pi = \{\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{x}, \mathtt{z}\}\}$.

4.2.3 Computational Complexity of ASP

In this section we discuss some aspects of computational complexity of ASP in which we consider the following issues:

i) Computation of answer sets (models) of a given program Π
ii) What search problems are expressible in ASP?

From *classical* complexity theory, issue (i) is basically a decision problem - *do there exist models of a program?* In the propositional case, Marek and Truszczyński (1991) showed that the problem of deciding whether a finite propositional logic program (which allows negation in the rule bodies but no disjunction in the rule heads) has answer set is NP-complete. Furthermore, allowing disjunctions in the rule heads, ASP can express the problem in the complexity class $\sum_2^P$ hence deciding if a disjunctive logic program has an answer set is $\sum_2^P$-complete. Hence compared with propositional satisfiability (SAT) which is NP-complete (Garey and Johnson, 1990), ASP is more powerful since problems which cannot be translated to SAT can be solved in polynomial time, unless $P = NP$ (Dantsin et al., 2001).

Regarding search problems, ASP can express all NP-search problems, that is, those solvable using a non-deterministic Turing machine in polynomial time, in such a way that the answer sets encode the solutions. In fact, each such problem (for example, finding some Hamiltonian cycle) is expressible by a fixed predicate program to which logical facts encoding a given problem instance (for example, a graph) are added. Again, additional constructs like disjunctive rules may increase the expressibility (Brewka et al., 2011). Detailed discussion of the com-

plexity classes relating to *AnsProlog* and its subclasses can be found in Baral (2003)

4.3 Reasoning about Specifications in ASP

Traces define the models (answer sets) that may be generated from an institution. In this section we are describing the trace program and the traces that the program may generate from our institutional specification. Also, we shall be describing how a query program may be used to constrain the traces such that we get the answer sets which satisfies our intended queries.

4.3.1 The Trace Program

The *AnsProlog* program $\Pi_{\mathcal{I}}^{base(n)}$ produced from the translation of an institution $\mathcal{I}$ does very little in the sense that it produces a single answer set which contains only the event facts provided in the program. For more useful results, information on occurrences of exogenous events need to be added and also provision of time line for the institution is required. This is achieved by defining a trace program. This provides sequences of exogenous events that are in turn interpreted by the base program to give a model. The types of trace programs we are presenting here are those that generate single traces and those that generate all possible traces of events up to length n.

When we only wish to determine the model of $\mathcal{I}$ over a single finite trace of a known length, we define a single trace program. For instance when we already know the events that have occurred and the order in which they occurred, single traces can be used to find the sequence of institutional events and states over that trace. The single trace program is defined as follows;

Definition 12. $\Pi_{\mathcal{I}}^{one(n)}$ *is an AnsProlog single trace program of length n for an institution* $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{G}, \Delta \rangle$ *made up of rules of the form:*

$$\texttt{observed}(\texttt{e}_\texttt{i}, \texttt{t}_\texttt{i}) \quad 0 \leq i < n, e_i \in \mathcal{E}_{ex}$$

such that only one rule is defined for each time instant $t_i : 0 \leq i < n$.

Following from Definition 12, we shall use the notation $\Pi_{\mathcal{I},tr}^{base(n)}$ to denote the single trace program corresponding to an ordered trace tr.

We assert that we can obtain a model of an institution by combining the trace information from the single trace program with the state transitions information encoded in $\Pi_{\mathcal{I}}^{base(n)}$. This will mean that the union of a single program trace and $\Pi_{\mathcal{I}}^{base(n)}$ results in a *stratified* program in which three distinct strata may be recognised for a given instant t_i thus; i.) fluents that are true at instant t_i, ii.) exogenous events that occur and institutional events that are generated between instants t_i and t_{i+1}, and iii.) the effects of those events.

Definition 13. *A partition $\pi_0, \ldots, \pi_k$ of the set of all predicate symbols of an AnsProlog program Π is a stratification of Π, if for any rule of the type $A_0 \leftarrow A_1, \ldots, A_m, \textbf{not } A_{m+1}, \ldots, \textbf{not } A_n$, and for any $p \in \pi_s, 0 \leq s \leq k$ if $A_0 \in atoms(p)$, then:*

(a) for every $1 \leq i \leq m$ there is q and $j \leq s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$, and

(b) for every $m+1 \leq i \leq n$ there is q and $j < s$ such that $q \in \pi_j$ and $A_i \in atoms(q)$.

A program is called stratified if it has a stratification.

In order to perform analysis and verification, a single trace is not sufficient we therefore need to define a trace program that will generate all the traces of length n for a given institution $\mathcal{I}$. This is defined as follows;

Definition 14. *A trace program $\Pi_{\mathcal{I}}^{all(n)}$ of length n for an institution $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{G}, \Delta \rangle$ is an AnsProlog program made up of the rules:*

$$\{\texttt{observed}(\texttt{e}_{\texttt{ex}}, \texttt{t}_{\texttt{i}})\}. \quad 0 \leq i < n, e_{ex} \in \mathcal{E}_{ex} \tag{14.1}$$

$$\texttt{ev}(\texttt{t}_{\texttt{i}}) \leftarrow \texttt{observed}(\texttt{e}_{\texttt{ex}}, \texttt{t}_{\texttt{i}}) \quad 0 \leq i < n, e_{ex} \in \mathcal{E}_{ex} \tag{14.2}$$

$$\leftarrow \textbf{not } \texttt{ev}(\texttt{t}_{\texttt{i}}) \qquad 0 \leq i < n \tag{14.3}$$

Rule 14.1 allows for any exogenous event $e_{ex} \in \mathcal{E}_{ex}$ to be non-deterministically

chosen as an observed event at any time instant. Also, the rules 14.2 and 14.3 ensures that at each instant an exogenous event must be observed and that at all instants at least one exogenous event must be observed. Therefore a trace program $\Pi_{\mathcal{I}}^{all(n)}$ generates as its answer set all possible combinations of n exogenous events together with the instant and `ev` facts provided. These are referred to as traces since they are time bound.

The generation of all traces is necessary for adequately handling reasoning problems since all the possible models of an institution are defined for a given period of time. However, in analysis and verification cases, we may for instance wish to simply find only traces which answer a particular question. Also, we may wish to determine the existence or non existence of answer sets for a particular question. To achieve this we need to constrain the traces generated by the trace program in combination with the base program. This is done by a *query program* defined as follows:

Definition 15 (query program)**.** *For a given base program $\Pi_{\mathcal{I}}^{base(n)}$ and trace program $\Pi_{\mathcal{I}}^{all(n)}$ of an institution $\mathcal{I}$, a program $\Pi_{\mathcal{I}}^{qry(n)}$ is a query program iff:*

$$Head(\Pi_{\mathcal{I}}^{qry(n)}) \cap Lit(\Pi_{\mathcal{I}}^{base(n)}) = \emptyset \tag{15.1}$$

$$Head(\Pi_{\mathcal{I}}^{qry(n)}) \cap Lit(\Pi_{\mathcal{I}}^{all(n)}) = \emptyset \tag{15.2}$$

that is to say that for both trace and base programs, all atoms which appear in the body of rules are disjoint with all atoms obtainable from the rules in a query program (i.e. those that appear in the head of the rules of the query program $\Pi_{\mathcal{I}}^{qry(n)}$).

When a trace program $\Pi_{\mathcal{I}}^{all(n)}$ is combined with the base program $\Pi_{\mathcal{I}}^{base(n)}$ for the institution $\mathcal{I}$ and a query program $\Pi_{\mathcal{I}}^{qry(n)}$, i.e $\Pi_{\mathcal{I}}^{all(n)} \cup \Pi_{\mathcal{I}}^{base(n)} \cup \Pi_{\mathcal{I}}^{qry(n)}$, we obtain an answer set which represents all possible traces and models of the institution $\mathcal{I}$ which are consistent with the query program $\Pi_{\mathcal{I}}^{qry(n)}$.

Our translated programs can also be used for reasoning tasks in the absence of complete information about the occurrence of events in a trace. This case would mean limiting answer sets to only those which are consistent with the partial

ordering of these events. We illustrate this with an example in which we assume knowledge of exactly which events have occurred and that events with same signature[1] only occur once. This is captured in the definition that follows;

Definition 16. *For a given set of exogenous events $E \subseteq \mathcal{E}_{ex}^{\mathcal{I}}$ and a partial ordering $R \subseteq E \times E$ of these events with $e_i \prec e_j \in R, e_i \neq e_j$ and $n = |E|$, we define the query program $\Pi_{\mathcal{I}}^{\preceq(n)}$ by the following rules;*

$$\begin{aligned}
\texttt{before}(\texttt{I1},\texttt{I2}) \quad &\leftarrow \quad \texttt{next}(\texttt{I1},\texttt{I2}),\texttt{instant}(\texttt{I1}),\texttt{instant}(\texttt{I2}).\\
\texttt{before}(\texttt{I1},\texttt{I3}) \quad &\leftarrow \quad \texttt{before}(\texttt{I1},\texttt{I2}),\texttt{before}(\texttt{I2},\texttt{I3}),\\
&\qquad\quad \texttt{instant}(\texttt{I1}),\texttt{instant}(\texttt{I2}),\texttt{instant}(\texttt{I3}).
\end{aligned}$$

The rules in Definition 16 define a transitive ordering over instants such that for each pair of instants t_i, t_j where $0 \leq i < j < n$ an atom $\texttt{before}(\texttt{t}_\texttt{i},\texttt{t}_\texttt{j})$ applies.

For each event ordering $e_i \prec e_j \in R$ a constraint rule in $\Pi_{\mathcal{I}}^{\preceq(n)}$ of the form $\leftarrow \texttt{observed}(\texttt{e}_\texttt{i},\texttt{I1}),\texttt{observed}(\texttt{e}_\texttt{j},\texttt{I2}),\texttt{before}(\texttt{I2},\texttt{I1}),\texttt{instant}(\texttt{I1}),\texttt{instant}(\texttt{I2}).$ limits the answer sets of the program to those where event e_i occurs before event e_j.

The combination $\Pi_{\mathcal{I}}^{\preceq(n)} \cup \Pi_{\mathcal{I}}^{all(n)} \cup \Pi_{\mathcal{I}}^{base(n)}$ will produce answer sets that only include models of traces where the ordered relation specified by R holds. The application of these types of trace are found in the case of agent reasoning, where an agent has partial knowledge about the events that have occurred or the ordering of events in the world.

4.4 Other Reasoning Approaches

A number of formal methods exist on how to represent and analyse the dynamics of a given complex domain in terms of the actions which are possible in that domain and their effects. These methods attempt to capture different aspects,

[1] An AnsProlog program is a pair $\{\sigma, \Pi\}$ where σ is signature and Π is a finite set of statements (rules) about σ (Gelfond, 2008).

such as goals, actors and actions, responsibilities and constraints. A complete study of this field is far beyond the scope of this thesis, however we summarise two approaches based on first order logic (the Situation Calculus and the Event Calculus), and contrast these with a number of approaches based in propositional logic.

4.4.1 Situation Calculus

One of the earliest attempts at formalising a model for change and action for use in artificial intelligence is the situation calculus first proposed by McCarthy in McCarthy (1963) and formalized in Levesque et al. (1998) and Reiter (2001). It is primarily a first-order logic language aimed at representing dynamically changing worlds, in which actions performed by agents are responsible for all changes. A first-order term, known as a *situation* represents a possible world history which is a sequence of actions. The situation calculus therefore revolves around the definition of a set of fluents, which describe both the corresponding (hypothetical) state of world and the effects of the performance of actions upon that state. Hence situation fluents in the situation calculus are relative to particular situations and treated as propositions. For instance, the fluent $in(\texttt{Bob}, \texttt{Building}, \sigma)$ states that in situation σ, `bob` is in the `building`. Also a function of the form $\texttt{Poss}(\alpha, \sigma)$ can be used to express that the action α is possible or executable in situation σ. Likewise the function $\texttt{Holds}(\rho, \sigma)$ indicates that fluent ρ is true in situation σ.

Change in the situation calculus is expressed through a successor function which takes some action and situation and produces a new situation. A successor function is denoted by a binary function $do(\alpha, \sigma)$. It implies that a new situation is brought about by the execution of action α in situation σ. Relations and functions whose truth values vary from situation to situation are called relational and functional fluents, respectively. The main ingredients of the situation calculus formalism that provide a complete treatment of reasoning about action are *situations*, *actions*, and $fluents$.

For better understanding, we illustrate the situation calculus reasoning with a simple example:

Consider an action $Travel(\texttt{Bob}, \texttt{Y})$ denoting that Bob has travelled from location Y, and a Boolean fluent $At(\texttt{Bob}, \texttt{Y}, \sigma)$ denoting that Bob is at location Y in situation σ, the semantics of this action will be defined as follows:

$$Poss(\texttt{Travel}(\texttt{Bob}, \texttt{Y}), \sigma) \leftarrow At(\texttt{Bob}, \texttt{Y}, \sigma) \quad (16.1)$$

$$\neg At(\texttt{Bob}, \texttt{Y}, \sigma') \leftarrow \sigma' = do(\texttt{Travel}(\texttt{Bob}, \texttt{Y}), \sigma) \quad (16.2)$$

The first axiom (16.1) states that it is only possible for Bob to travel from location Y in situation σ, if Bob is at location Y in situation σ. The second axiom (16.2) states that if Bob does travel from Y in situation σ then Bob is not considered to be at location Y in the situation σ' which immediately follows σ.

A number of deficiencies were observed in this approach, notably the *frame problem* (McCarthy and Hayes, 1968) which stems from the fact that situation calculus operates on partial description of the world. The significance of this problem has led to several approaches which aim at solving the problem. These include the event calculus approach we discuss in the next section.

4.4.2 Event Calculus

The event calculus originally proposed by Kowalski and Sergot (1986) is another logic-based framework for representing and reasoning about actions that has attracted much attention as a solution to the frame problem of the situation calculus. The event calculus and situation calculus are similar in the sense that they can both be formalized by means of Horn clauses augmented with negation by failure. However, they differ in that while situation calculus deals with global states, event calculus deals with local events and time periods. The event calculus is able to express both how events (also called actions) may change the valuations of fluents, and how, based on the occurrence of events at an earlier time, the valuations of fluents may be determined. Time-points are used to axiomatise expressions in such a way that fluents are true if they have been initiated by an event occurrence at some earlier time-point. A central feature of the event calculus, called a *narrative*, is a possibly incomplete specification of a set of actual event occurrences. By tackling the frame problem using circumscription (McCarthy, 1980) and narrative, event calculus is capable of representing a variety of

$$Clipped(t_1, f, t_2) \stackrel{def}{\equiv} \exists a, t(Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Terminates(a, f, t)). \quad \text{(EC1)}$$

$$Declipped(t_1, f, t_2) \stackrel{def}{\equiv} \exists a, t(Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Terminates(a, f, t)). \quad \text{(EC2)}$$

$$HoldsAt(f, t_1) \leftarrow [Happens(a, t_1) \wedge Initiates(a, f, t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1, f, t_2)]. \quad \text{(EC3)}$$

$$\neg HoldsAt(f, t_1) \leftarrow [Happens(a, t_1) \wedge Terminates(a, f, t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1, f, t_2)]. \quad \text{(EC4)}$$

$$HoldsAt(f, t_2) \leftarrow [HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1, f, t_2)]. \quad \text{(EC5)}$$

$$\neg HoldsAt(f, t_2) \leftarrow [HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1, f, t_2)]. \quad \text{(EC6)}$$

Figure 4-2: Some axioms of Event Calculus

phenomena more naturally and also to perform non-monotonic reasoning, as described in many works, such as Mueller (2008); Shanahan (1999). Event calculus attempts to solve the frame problem by reifying when fluents hold in particular time points as seen in Figure 4-2 which presents axiomatised summary of the commonly used simplified event calculus as found in Shanahan (1999); Miller and Shanahan (2002). The predicates involved in these axioms are explained in Table 4.1.

Predicate	**Meaning**
$Clipped(t_1, f, t_2)$	causes a fluent f to start holding between time-points t_1 and t_2
$Declipped(t_1, f, t_2)$	causes a fluent f to cease holding between time-points t_1 and t_2
$Happens(a, t)$	event a happens at time-point t
$HoldsAt(f, t)$	fluent f holds at time point t
$Initiates(a, f, t)$	event a causes fluent f to start at time-point t
$Terminates(a, f, t)$	event a causes fluent f to stop at time-point t

Table 4.1: Commonly used Event Calculus Predicates

In addition to the event calculus *domain independent* axioms presented in Figure 4-2, the theory of event calculus also demands the definition of *domain dependent* axioms which specify the creation of the $Initiates$ and $Terminates$ fluents with

respect to given events. For example let us take the theory `"Depressing the switch button turns on the light, releasing the switch button turns off the light"`. Representing the depressing and releasing of the switch button with the events *Depress* and *Release* and when the light is turned on with the fluent *On*, the stated theory could be expressed as:

$$Initiates(a, f, t) \stackrel{def}{\equiv} [a = Depress, f = On, \neg HoldsAt(On, t)]$$
$$Terminates(a, f, t) \stackrel{def}{\equiv} [a = Release, f = On, HoldsAt(On, t)]$$

It is worth noting that event calculus assumes and enforces the common-sense law of inertia which implies that the truth value of a fluent is preserved from its initial time-point until it is terminated by some other event. When a fluent is not subject to the common-sense law of inertia its truth value is allowed to fluctuate in an arbitrary fashion and therefore may or may not be the same. A number of tools exist for reasoning using descriptions based on the logic. These range from the use of abduction (Denecker et al., 1992), theorem proving (Shanahan, 1997) to a model-based reasoning technique based on SAT-solvers (Mueller, 2004).

4.4.3 Action Languages

A number of approaches have been proposed for reasoning about actions (see Gelfond and Lifschitz (1993); Baral (2010); Strass and Thielscher (2012)). Common approaches that model actions are based on the notion of state change. Actions are generally considered instantaneous and defined as functions from one state to another by means of predefined conditions. Reasoning about actions is desirable in many ways; *i.)* it helps us to predict if a sequence of actions is indeed going to achieve some desired goal; *ii.)* it allows us to plan or come up with a sequence of actions that would achieve a particular goal and maintain particular trajectories; *iii.)* it allows us to explain observations in terms of what actions may have taken place; and *iv.)* it allows us to diagnose faults in a system in terms of finding what actions may have taken place to result in the faults. When actions have non-deterministic effects reasoning about actions is needed to verify policies and come up with policies to achieve goals and maintain desired trajectories. Thus, reasoning about actions is an important topic in Computer Science in general

and in AI in particular. It has also served as a benchmark domain for evaluating knowledge representation languages (Baral, 2010).

The situation calculus and event calculus, being first and second order logic formalisms respectively provide expressive means for modelling change and time in dynamic systems. However these systems come at a computational cost, and tools for reasoning with both of these systems make a broad class of reasoning problems intractable presently. These have led to researchers proposing a number of proposition logic-based approaches including the action language $\mathcal{A}$ (Gelfond and Lifschitz, 1993).

The semantics of the action languages are based on the causal theories in McCain and Turner (1997), which distinguish between the claim that a formula is true and the stronger claim that there is a cause for it to be true. Based on this, the first high-level action language $\mathcal{A}$ was developed. This language only supports expressions of the form "α **causes** β **if** γ" where α is an action name, β a literal and γ a conjunction of literals (possibly empty). The action language $\mathcal{C}$ (Giunchiglia and Lifschitz, 1998) extends the language $\mathcal{A}$ by providing additional language expressions, besides direct effects of actions, such as dependencies between fluents.

The language $\mathcal{C}+$ proposed by Giunchiglia, Lee and Lifschitz in Giunchiglia et al. (2001) is an extension to the languages $\mathcal{C}$ and $\mathcal{A}$ with the inclusion of *dynamic laws* and *multi-valued fluents*. In this sense, when determining the value of fluents, $\mathcal{C}+$ permits the definition of a class of fluents called *dynamic fluents*. In any given state, the value of these fluents is based on an expression evaluated over the other fluents in the current state. This is unlike in $\mathcal{A}$ where the value of a fluent is dependent only on the previous state and actions which have occurred. This addition makes the representation of a number of properties in the language simple and more succinct. Also in contrast with the Boolean fluents in $\mathcal{A}$ which may only be true or false in a given state, multi-valued fluents may take on a number of values.

In terms of model size, given a state property which may take on exactly one of N values, the multi-valued fluents in $\mathcal{C}$ reduces the number of possible state-fluents to exactly N leading to a corresponding decrease in model size. This

contrasts with $\mathcal{A}$ where each of the N values must be represented as a Boolean fluent leading to 2^N possible combinations of the fluent values. Another contrast also deals with the underlying semantics. $\mathcal{A}$ has semantics based on answer set programs while the semantics for $\mathcal{C}+$ is based on the logic of causal theories which defines the circumstances under which the valuation of a formula can be considered to be caused.

4.4.4 Model Checking

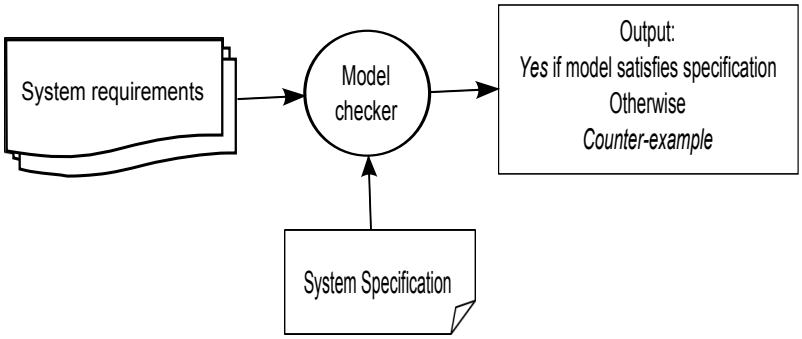


Figure 4-3: An Overview of the Model Checking Approach

In formal logic, model checking designates the problem of determining whether a formula ϕ evaluates to true or false in an interpretation $\mathcal{M}$, written $\mathcal{M} \models \phi$. For example, $\mathcal{M}$ might represent a knowledge base or a system and ϕ could be a query of which we wish to determine if it is implied by the knowledge in the base or a formula representing the correctness of the system being verified. We are then interested in finding efficient algorithms for determining whether $\mathcal{M} \models \phi$ holds (Merz, 2008). As described in Figure 4-3, the inputs of the model checking tool consist of the system description and the property to be verified. The tool either outputs a Yes if the system description (model) satisfies the system property, otherwise a $counter - example$ is presented to the user.

Model checking uses an algorithmic technique to formally verify properties of a finite state system. It combines temporal logic formula which is used to express the property to be verified and state transition system which captures the model to be checked. The states of the system and actions or events that cause the system to change states are represented by the nodes and edges respectively of the state transition system. The model checking procedure aims at satisfying the

expression $(\mathcal{M}, s \models \phi)$, where $\mathcal{M}$ denotes the model, s the initial state and ϕ the property being checked. Informally, this can be stated as, given a model $\mathcal{M}$ and an initial state s it is possible to prove the property ϕ.

Typically the correctness properties for model checking are expressed either in computational tree logic (CTL) or linear time logic (LTL). Model checking tools such as SPIN (Holzmann, 1997) for LTL and NuSMV (Cimatti et al., 1999) for CTL have been developed and used for verifying systems containing large number of states. These tools verify a given property by means of state exploration and symbolic model checking. They take the user specified model, initial state and the property to be verified and attempt to check if the model satisfies the property and responding with either a "yes" or a counter-example which is a sequence of event/action transitions through the state model that cause the property to be false. This shows situations where the system behaves in such a way that the property is violated, and the absence of counterexamples indicates that the modelled system satisfies the property (Bandara et al., 2003).

Although model checking can be applied to much larger state spaces than those which can be studied using ASP, queries are limited to those which can be expressed in the temporal logic used by the underlying model checker. In the case of CTL for instance we are limited to formulae which are quantified over all future paths making some queries impossible to specify. As such temporal logic model checking may be seen as a complementary method for checking properties that require the full investigation of the state space, but not do not require the expressive power of ASP.

Another type of automated model checking is the bounded temporal logic model checking (Biere et al., 2003). Here, the correctness properties that can be investigated are limited to those which can be detected in a finite trace of the system. Unlike symbolic model checking, bounded model checking may be conducted using more expressive logics such as CTL* (Visser and Barringer, 2000), relaxing the expressiveness constraints of logics such as CTL and LTL. Typically, bounded logic model checking systems are comparable to ASP in their approach to modelling investigations.

While model checking has found great applications especially in hardware verifi-

cation (Burch et al., 1990) and recently software verification (Bérard et al., 2010), in the context of security requirements analysis, model checking has a number of significant disadvantages. These include the inability to deal with partial specification of initial states and to check static properties of the system.

Verification in model checking approaches as presented by SPIN and NuSMV are concerned with the problem of reachability of states and the satisfiability of constraints. ASP on the other hand is concerned with the problem of finding stable models of specified rules. This means that with ASP, models which satisfy specified constraints can be generated.

4.5 Institutional framework to ASP Translation

As we have pointed out in Section 3.6 on page 47, the institutional framework allow us to express the specifications of an organisational "workflow" and the elicited security requirements. We also presented the computational framework which allows us to formalise and reason about these specifications for the purpose of analysis which is the objective of our work. The idea lies in the fact that if an institution $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{G}, \Delta \rangle$ is represented as an *AnsProlog* program Π, then the answer sets to this program corresponds to the traces of the institution.

For us to achieve this goal of representing our institutional specifications in *AnsProlog*, it is necessary to map our institutional framework definition $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{G}, \Delta \rangle$ and all its components into answer set programs such that the answer set programs described by this translation model the semantics of the target institution. By expressing verification problems as queries over these programs, it becomes possible to determine the presence or absence of desirable properties of the original institution specification, through the presence or absence of answer sets to queries over the institutional program. We shall proceed to explain this translation as presented in Cliffe et al. (2007).

For a given institution $\mathcal{I}$, the mapping follows a definition of three components which together result in the *AnsProlog* program $\Pi_{\mathcal{I}}$. These components are: time component Π^{n}, base component Π^{base}, and an institution specific component $\Pi^{*}_{\mathcal{I}}$. Together, these three components generate the answer set program $\Pi^{base(n)}_{\mathcal{I}}$ which is capable of providing all the ordered traces of length n for the institution $\mathcal{I}$.

We shall be explaining the mapping of the institutional specifications with ASP through these components.

4.5.1 Translation Based on Components

In doing the mapping of the institution $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{G}, \Delta \rangle$ to *AnsProlog*, the atoms presented in Table 4.2are used. We now explain the role and usage of these atoms

Atom	Meaning
`ifluent(P)`	identify fluents
`nifluent(P)`	identify non-inertial fluents
`evtype(E,T)`	describe the type of an event
`event(E)`	denote events
`instant(I)`	denote time instants
`final(I)`	last time instant
`next(I1,I2)`	time ordering, `I1` comes before `I2`
`occurred(E,I)`	the event `E` happened at time instant `I`
`observed(E,I)`	the event `E` was observed at time instant `I`
`holdsat(P,I)`	the institutional fluent `P` holds at time instant `I`
`initiated(P,I)`	fluent `P` initiated at time instant `I`
`terminated(P,I)`	fluent `P` terminated at time instant `I`

Table 4.2: Atoms used for mapping institutions to *AnsProlog*

in the translation starting with the time component Π^n.

Time component Π^n

This defines the predicates for time instants of length n such that $t_i : 0 \leq i \leq n$ and is responsible for generating a single observed event at every time instant. We noted in Section 3.6.1.2 that the general notion of time in an institutional framework is based on assumption that time consists of a number of ordered time instants, without any particular regard for the actual duration between two of these instants. Time instants are therefore taken to represent the state of an institution at a given time such that each time instant corresponds to one possible institutional state. Events are considered to occur between the time instants so that event occurring at time $\mathtt{t_i}$ is considered to occur in the real world between the time instants $\mathtt{t_i}$ and $\mathtt{t_{i+1}}$. *AnsProlog* definition of time is given by the

following facts:

$$0 < k < n \quad : \quad \texttt{instant}(i_k). \tag{16.3}$$

$$0 < k < n-1 \quad : \quad \texttt{next}(i_k, i_{k+1}). \tag{16.4}$$

$$\texttt{final}(i_k). \tag{16.5}$$

where (16.3) defines all time instants available in the institution while (16.4) specifies the ordering of times between two instants as necessary for transition from one state to another. The fact in (16.5) represents the final state since it is not possible to have an observable event occurring at the final time instant.

Base component Π^{base}

Following the presentation of time interpretation, we now present the Π^{base} component which consist of general rules that hold for any institution. These rules are responsible for the occurrence of observed events, handling of fluent inertia (i.e. the transition of the states) and dealing with the generation of violation events which could be due to prohibited events (i.e not permitted events) or unsatisfied obligations. The translation rules for the institution base program are shown in Figure 4-4.

The first rule (16.6) has to do with exogenous events $\mathcal{E}_{ex}$ and ensures that each observed exogenous event `E` is marked as occurred, given that all observable events are valid events (due to the autonomy of the actors in the institutional framework, any exogenous event can occur since they are not controllable by the institution). Rules (16.7) deals with standard inertia and uses negation as failure. The rule ensures that any fluent which is valid at an instant `I1` `(holdsat(I1))` not terminated in this state `(not terminated(P,I1))` still remains valid in the next state `(holdsat(P,I2))`. The atoms `next(I1,I2)` and `instant(I1,I2)` handle the generation of the next time instant (state) and its grounding. Focusing on initiation rules, rule (16.8) makes sure that fluents which are initiated at an instant `(initiated(P,I1))` become valid in the next state `(holdsat(P,I2))`. Events that are not permitted are expected to generate a violation. This is handled by rule (16.9) which ensures that violations are generated `(occurred(viol(E),I))` whenever an event happens `(occurred(E,I))`

$$\texttt{occurred}(\texttt{E},\texttt{I}) \;\texttt{:-}\; \texttt{observed}(\texttt{E},\texttt{I}). \tag{16.6}$$

$$\begin{aligned}\texttt{holdsat}(\texttt{P},\texttt{I2}) \;\texttt{:-}\;\; & \texttt{holdsat}(\texttt{P},\texttt{I1}),\\ & \texttt{not terminated}(\texttt{P},\texttt{I1}),\\ & \texttt{next}(\texttt{I1},\texttt{I2}),\texttt{instant}(\texttt{I1},\texttt{I2}),\\ & \texttt{ifluent}(\texttt{P}).\end{aligned} \tag{16.7}$$

$$\begin{aligned}\texttt{holdsat}(\texttt{P},\texttt{I2}) \;\texttt{:-}\;\; & \texttt{initiated}(\texttt{P},\texttt{I1}),\texttt{next}(\texttt{I1},\texttt{I2}),\\ & \texttt{instant}(\texttt{I1},\texttt{I2}),\texttt{ifluent}(\texttt{P}).\end{aligned} \tag{16.8}$$

$$\begin{aligned}\texttt{occurred}(\texttt{viol}(\texttt{E}),\texttt{I}) \;\texttt{:-}\;\; & \texttt{occurred}(\texttt{E},\texttt{I}),\\ & \texttt{not holdsat}(\texttt{perm}(\texttt{E}),\texttt{I}),\\ & \texttt{event}(\texttt{E}),\texttt{event}(\texttt{viol}(\texttt{E})),\\ & \texttt{instant}(\texttt{I}).\end{aligned} \tag{16.9}$$

$$\begin{aligned}\texttt{occurred}(\texttt{V},\texttt{I}) \;\texttt{:-}\;\; & \texttt{holdsat}(\texttt{obl}(\texttt{E},\texttt{D},\texttt{V}),\texttt{I}),\\ & \texttt{occurred}(\texttt{D},\texttt{I}),\texttt{event}(\texttt{E},\texttt{D},\texttt{V}),\\ & \texttt{instant}(\texttt{I}).\end{aligned} \tag{16.10}$$

$$\begin{aligned}\texttt{terminated}(\texttt{obl}(\texttt{E},\texttt{D},\texttt{V}),\texttt{I}) \;\texttt{:-}\;\; & \texttt{occurred}(\texttt{E},\texttt{I}),\\ & \texttt{holdsat}(\texttt{obl}(\texttt{E},\texttt{D},\texttt{V}),\texttt{I}),\\ & \texttt{event}(\texttt{E},\texttt{D},\texttt{V}),\texttt{instant}(\texttt{I}).\end{aligned} \tag{16.11}$$

$$\begin{aligned}\texttt{terminated}(\texttt{obl}(\texttt{E},\texttt{D},\texttt{V}),\texttt{I}) \;\texttt{:-}\;\; & \texttt{occurred}(\texttt{D},\texttt{I}),\\ & \texttt{holdsat}(\texttt{obl}(\texttt{E},\texttt{D},\texttt{V}),\texttt{I}),\\ & \texttt{event}(\texttt{E},\texttt{D},\texttt{V}),\texttt{instant}(\texttt{I}).\end{aligned} \tag{16.12}$$

Figure 4-4: The Rules for Π^{base} Translation

$$\{\texttt{observable(E, I)}\} \;\texttt{:-}\; \texttt{evtype(E, obs), event(E), instant(I), not final(I).} \quad (16.13)$$

$$\texttt{ev(E, I)} \;\texttt{:-}\; \texttt{observed(E, I), event(E), instant(I).} \quad (16.14)$$

$$\texttt{:-}\; \texttt{not ev(I), instant(I), not final(I).} \quad (16.15)$$

$$\texttt{:-}\; \texttt{observed(E1, I), observed(E2, I), E1! = E2, instant(I), event(E1), event(E2).} \quad (16.16)$$

Figure 4-5: The Rules for handling Observable Traces

in a state for which no permission exists `(not holdsat(perm(E),I))` for that event. Obligation issues are handled by rules (16.10) - (16.12). Rule (16.10) handles the need for a violation to be raised `(occurred(V,I))` whenever the deadline `D` for an obligation expires `(occurred(D,I))` at the same instant that the obligation is still valid `holdsat(obl (E,D,V),I)`. An obligation can be terminated `(terminated(obl(E,D,V),I))` by the occurrence `(occurred(E,I))` of the obliged event. This is ensured by rule (16.11). Also the occurrence of the deadline event `(occurred(D,I))` associated to an obligation can cause the termination of the obligation.

The semantics of the institutional framework is described as change of states brought about by occurrence of institutional events $\mathcal{E}_{inst}$ triggered by a sequence of exogenous (real world) events $\mathcal{E}_{ex}$ taking place. Each of the sequence of exogenous events in the institution represents a single possible model for the interpretation of the institution in *AnsProlog*. Therefore, in our specification of the computational model of an institution we are particularly interested only in models that have had any exogenous event $\mathcal{E}_{ex}$ occurring. This means that we are only interested in models that have traces containing observable events. Because of this, there is a need to constrain the answer sets or models by adding a set of rules to the Π^{base} rules that would ensure we get the kind of models we are interested in. The rules in Figure 4-5 are therefore added to the base rules in Figure 4-4.

Here, the generation of `observed(E,I)` atoms is handled by rule (16.13) which en-

$$EX(x_1 \wedge x_2 \wedge \ldots \wedge x_n, t_i) \stackrel{def}{\equiv} EX(x_1, t_i), EX(x_2, t_i), \ldots, EX(x_n, t_i) \quad (16.17)$$

$$EX(\neg f, t_i) \stackrel{def}{\equiv} \texttt{not}\ EX(f, t_i) \quad (16.18)$$

$$EX(f, t_i) \stackrel{def}{\equiv} \texttt{holdsat(f,t}_\texttt{i}\texttt{)} \quad (16.19)$$

Figure 4-6: Translation of the Condition Statement

sures that for each combination of observable (`evtype(E,obs)`) event (`event(E)`) that is non-final (`not final(I)`) at time instant `instant(I)`, an {`observed(E,I)`} choice is generated. This indicates that it is possible to either use the `observed(E,I)` atom or not. For each choice, rule (16.14) generates an event `ev(E,I)` which is the used by rule (16.15) to restrict the answer sets to only those that have at least one observable event ($\mathcal{E}_{ex}$) at each time step. In the same way also, rule (16.16) ensures that each answer set has at most one $\mathcal{E}_{ex}$ at every time instant.

Institution specific component $\Pi^*_{\mathcal{I}}$

This component specifies all the components that are specific to the institution being modelled.

In order to account for the effect of institutional rules over a specific set of states, in Section 3.6.2 we defined $\mathcal{X}$ as the set of state formulae over a given set of institutional fluent properties $\mathcal{F}$ which may be true while the institution is active. With this, an auxiliary function EX such that given an expression $\phi \in \mathcal{X}$ at time instant t_i, ϕ can be translated into ASP atoms using the term $EX(\phi, t_i)$ as presented in Figure 4-6.

For example, the condition $\neg x, y$ at time instant t_i would be translated into the following sequence of extended *AnsProlog* literals:

$$\texttt{not holdsat(x,t}_\texttt{i}\texttt{)}, \texttt{holdsat(y,t}_\texttt{i}\texttt{)}.$$

With these rules in place, the rules for translating $\Pi^*_{\mathcal{I}}$ becomes the rules presented in Figure 4-7.

Rule (16.20) allows fluents to be encoded as facts `ifluent(p)` which facili-

$$
\begin{aligned}
p \in \mathcal{F} &\Leftrightarrow \texttt{ifluent(p)}. && (16.20)\\
e \in \mathcal{E} &\Leftrightarrow \texttt{event(e)}. && (16.21)\\
e \in \mathcal{E}_{ex} &\Leftrightarrow \texttt{evtype(e,obs)}. && (16.22)\\
e \in \mathcal{E}_{act} &\Leftrightarrow \texttt{evtype(e,act)}. && (16.23)\\
e \in \mathcal{E}_{viol} &\Leftrightarrow \texttt{evtype(e,viol)}. && (16.24)\\
e \in \mathcal{E}_{+} &\Leftrightarrow \texttt{evtype(e,cr)}. && (16.25)\\
\mathcal{C}^{\uparrow}(\phi, e) = P &\Leftrightarrow \forall p \in P \cdot \texttt{initiated(p,I)} :\!- \texttt{occurred(e,I)}, EX(\phi, I). && (16.26)\\
\mathcal{C}^{\downarrow}(\phi, e) = P &\Leftrightarrow \forall p \in P \cdot \texttt{terminated(p,I)} :\!- \texttt{occurred(e,I)}, EX(\phi, I). && (16.27)\\
\mathcal{G}(\phi, e) = E &\Leftrightarrow \forall g \in E \cdot \texttt{occurred(g,I)} :\!- \texttt{occurred(e,I)}, \texttt{holdsat(pow(e),I)}, EX(\phi, I). && (16.28)\\
p \in \Delta &\Leftrightarrow \texttt{holdsat(p,}\texttt{i}_0\texttt{)}. && (16.29)
\end{aligned}
$$

Figure 4-7: Translation Rules for $\Pi^*_{\mathcal{I}}$

tates grounding in the *AnsProlog* program. Rules (16.21) - (16.24) deals with events such that each event $e \in \mathcal{E}$ in the institution generates two facts - a fact `event(e)`which denotes an event (rule 16.21), and a fact `evtype(e,X)` which denotes the event type `X` $\in$ `obs, act, viol, cr` (rules 16.22 - 16.25) representing observable events, institutional actions, violation and creation events (see Section 3.6.2). The rules for consequence generation are provided for by rules (16.26) and (16.27). These rules are responsible for ensuring that whenever a fluent needs to be initiated or terminated a rule will be created appropriately. The structure of this rule is such that the initiation/termination atom is in the head of the rule `(initiated(p,I)/terminated(p,I))` while the occurrence of the responsible event `(occurred(e,I))` and the conditions on the state `(EX(X,I)` form the body of the rule. Rule 16.28 handles the event generation and for each event generated, the produced rule contains the occurrence of the responsible event `(occurred(e,I))` together with the power to execute this event `(holdsat(pow(e),I))` and the conditional rules `(EX(X,I)` in the body. The occurrence of the generated event `(occurred(g,I))` is presented as the head of the rule. Lastly, rule 16.29 encodes the initial state Δ of the institution in which each fluent $p \in \Delta$ is translated into a fact `holdsat(p,`$\texttt{i}_0$`)`.

4.5.2 Summary of the Translations

We have so far presented the translations for the three components $\Pi^*_{\mathcal{I}}$, Π^n, and Π^{base} which together make up the answer set program $\Pi^{base(n)}_{\mathcal{I}}$ corresponding to an institutional specification $\mathcal{I} = \langle \mathcal{E}, \mathcal{F}, \mathcal{G}, \Delta \rangle$. In Table 4.3 we present a summary of the semantics for translation of an institution $\mathcal{I}$ into a corresponding *Ansprolog* program $\Pi^{base(n)}_{\mathcal{I}}$. With this, we are able to generate an answer set program that is capable of providing all ordered traces of length n for the institution $\mathcal{I}$. Where necessary, we can also add additional rules to the program that would restrict the answer sets to those which fulfil certain desired properties being investigated. In the next section, we present a high-level action language InstAL through which we shall be modelling our institutions.

Institution specification	*AnsProlog* **Program Translation**
$f \in \mathcal{F}^*$	$\texttt{holdsat}(\texttt{f}, \texttt{t}_{\texttt{i+1}}) \leftarrow \texttt{initiated}(\texttt{f}, \texttt{t}_\texttt{i})$.
$f \in \mathcal{F}^*$	$\texttt{holdsat}(\texttt{f}, \texttt{t}_{\texttt{i+1}}) \leftarrow \texttt{holdsat}(\texttt{f}, \texttt{t}_\texttt{i}),$ **not** $\texttt{terminated}(\texttt{f}, \texttt{t}_\texttt{i})$.
$e_{ex} \in \mathcal{E}_{ex} - \mathcal{E}_+$	$\texttt{occurred}(\texttt{e}_\texttt{ex}, \texttt{t}_\texttt{i}) \leftarrow \texttt{observed}(\texttt{e}_\texttt{ex}, \texttt{t}_\texttt{i}),$ $\texttt{holdsat}(\texttt{live}, \texttt{t}_\texttt{i})$.
$e_{cr} \in \mathcal{E}_+$	$\texttt{occurred}(\texttt{e}_\texttt{cr}, \texttt{t}_\texttt{i}) \leftarrow \texttt{observed}(\texttt{e}_\texttt{cr}, \texttt{t}_\texttt{i})$.
$e_1 \in \mathcal{E}, \phi \in \mathcal{X}, e_2 \in \mathcal{G}(\phi, e_1), e_2 \in \mathcal{E}_{act}$	$\texttt{occurred}(\texttt{e}_2, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}_1, \texttt{t}_\texttt{i}), \texttt{EX}(\phi, \texttt{t}_\texttt{i}),$ $\texttt{holdsat}(\texttt{pow}(\texttt{e}_2), \texttt{t}_\texttt{i})$.
$e_1 \in \mathcal{E}, \phi \in \mathcal{X}, e_2 \in \mathcal{G}(\phi, e_1), e_2 \in \mathcal{E}_{viol}$	$\texttt{occurred}(\texttt{e}_2, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}_1, \texttt{t}_\texttt{i}), \texttt{EX}(\phi, \texttt{t}_\texttt{i})$.
$e \in (\mathcal{E}_{act} - \mathcal{E}_-) \cup (\mathcal{E}_{ex} - \mathcal{E}_+)$	$\texttt{occurred}(\texttt{viol}(\texttt{e}), \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i}),$ **not** $\texttt{holdsat}(\texttt{perm}(\texttt{e}), \texttt{t}_\texttt{i})$.
$\text{obl}(e, d, v) \in \mathcal{O}$	$\texttt{occurred}(\texttt{v}, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{d}, \texttt{t}_\texttt{i}),$ $\texttt{holdsat}(\texttt{obl}(\texttt{e}, \texttt{d}, \texttt{v}), \texttt{t}_\texttt{i})$.
$f \in \Delta \cup live, e_{cr} \in \mathcal{E}_+$	$\texttt{initiated}(\texttt{f}, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}_\texttt{cr}, \texttt{t}_\texttt{i}),$ **not** $\texttt{holdsat}(\texttt{live}, \texttt{t}_\texttt{i}),$ **not** $\texttt{dissolved}(\texttt{inst}, \texttt{t}_\texttt{i})$.
$e \in \mathcal{E}, \phi \in \mathcal{X}, f \in \mathcal{C}^{\uparrow}(\phi, e)$	$\texttt{initiated}(\texttt{f}, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i}), \texttt{EX}(\texttt{x}, \texttt{t}_\texttt{i}),$ **not** $\texttt{dissolved}(\texttt{inst}, \texttt{t}_\texttt{i})$.
$e \in \mathcal{E}, \phi \in \mathcal{X}, f \in \mathcal{C}^{\downarrow}(\phi, e)$	$\texttt{terminated}(\texttt{f}, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i}), \texttt{EX}(\texttt{x}, \texttt{t}_\texttt{i})$.
$\text{obl}(e, d, v) \in \mathcal{O}$	$\texttt{terminated}(\texttt{obl}(\texttt{e}, \texttt{d}, \texttt{v}), \texttt{t}_\texttt{i})\texttt{t} \leftarrow \texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i})$.
$\text{obl}(e, d, v) \in \mathcal{O}$	$\texttt{terminated}(\texttt{obl}(\texttt{e}, \texttt{d}, \texttt{v}), \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{d}, \texttt{t}_\texttt{i})$.
$f \in \mathcal{F}^*$	$\texttt{terminated}(\texttt{f}, \texttt{t}_\texttt{i}) \leftarrow \texttt{dissolved}(\texttt{inst}, \texttt{t}_\texttt{i})$.
$e \in \mathcal{E}_-$	$\texttt{dissolved}(\texttt{inst}, \texttt{t}_\texttt{i}) \leftarrow \texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i})$.

Table 4.3: *AnsProlog* Program Translation for all time instants

4.6 The Action Language InstAL

In the last section we show that the formal model of an institution can be translated to *AnsProlog* program such that the solutions of the program, known as answer sets of the program, defined through the answer set semantics (Baral, 2003) correspond to the traces of the institutional framework. However this translation would require an appreciable knowledge of *AnsProlog* formalization and syntax. Therefore in order to make this translation easier, a layer of abstraction on top of *AnsProlog* was provided in Cliffe et al. (2006) through the domain-specific action language called InstAL. This language InstAL describes the various components of the institution using a semi-natural language. In this section we present the features of this language as we apply it to various aspects of our models. Figure 4-8 gives an overview of the components and the process of instAL translation process. An InstAL reasoning task consists of the following components:

i.) An InstAL *institution specification* which describes an institution that should be processed.

ii.) A *domain definition* that contains grounding information for aspects of the institution description. This provides the domains for *types* and any static properties referenced in the institution definition.

iii.) A *trace program* which defines the set traces of exogenous events that should be investigated.

iv.) A *query program* which describes the desired property which should be validated by the InstAL reasoning tool.

4.6.1 Specifying Institutions in InstAL

While we do not intend to present the details of InstAL here since this can be found in Cliffe (2007) we, for completeness, describe the features of InstAL that are useful for our security requirements specification and how the various institutional components described in Section 3.6.2 may be presented in InstAL.

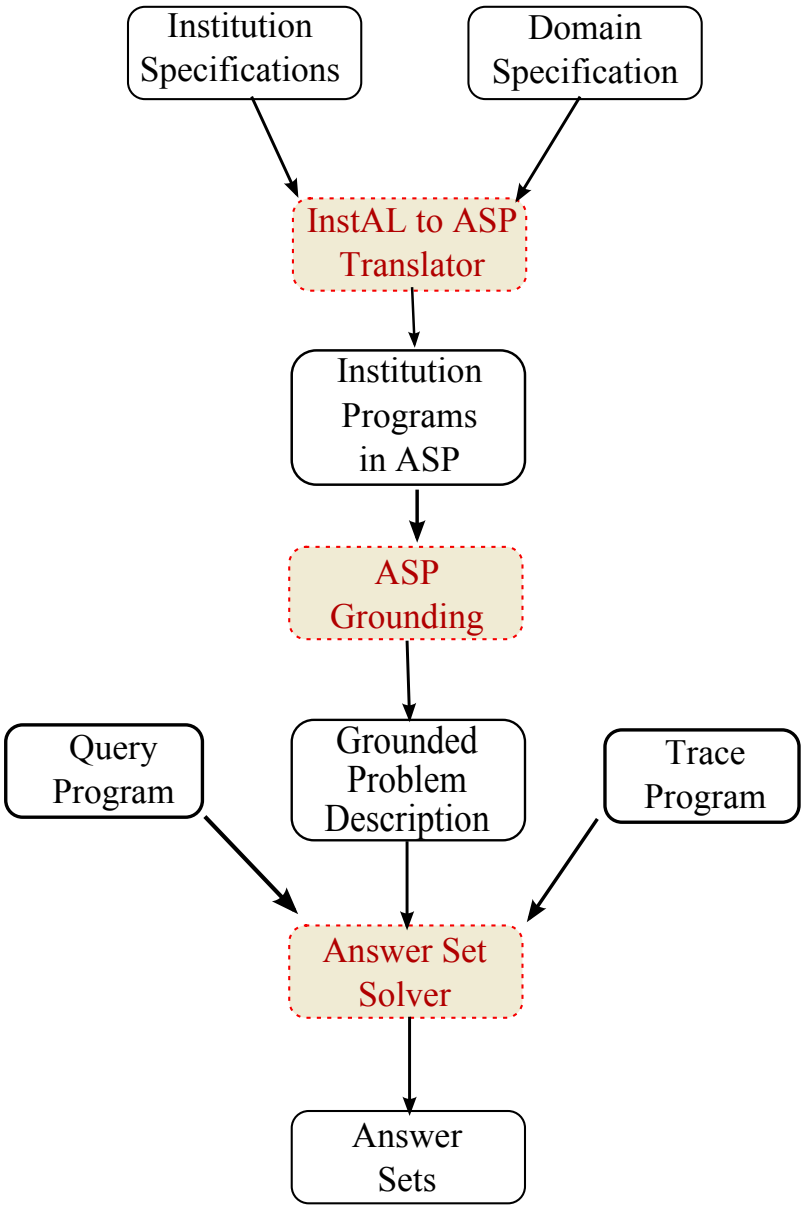


Figure 4-8: Overview of Inst*AL* translation process (Cliffe, 2007)

Domain Specification

The domain definition consist of statements of the form:

```
type-identifier:type-value(s)
```

where `type-identifier` denotes the variable declared as `type` in the associated institutional specification while `type-value(s)` is the list of atoms (constants)

separated by spaces, needed for the grounding of the variables.

For instance, the description:

```
Lecturer: lecturer
Student: student
ExOffice: eo
Head: hod
Paper: paper
```

specifies the content of our domain file for an exams scenario. For example the variables `ExamOffice` and `Head` of the underlying institution specification would be grounded with the values `examOfficer` and `hod` respectively. A domain description file should contain complete information necessary for each of the variables that should be grounded. This separation of domain information from the model description allows us to make changes to the values without having to change the institutional specifications.

Institutional Specification

The institutional specification is made up of declarations (type, events, and fluents), rule descriptions (causal, generation, and initiation rules). We shall explain how these declarations and rules may be specified in InstAL here.

Type Declarations: The InstAL type declaration takes the form:

`type` *identifier*;

where the `type` *identifier* declares a type such as:

```
type Lecturer;
type Student;
type ExOffice;
type Head;
type Paper;
```

As presented earlier, a domain file specifies the acceptable values for each declared type in the form `type-identifier:type-value(s)` with the values separated by spaces. An example of such a specification of grounding values in the domain file for our description is `Agent: eo hod`. InstAL will then substitute those values

(`eo` and `hod`) whenever the respective type `Agent` is specified for the events and fluents.

InstAL translation tool automatically generates three internally-defined types using the specification being processed in addition to the user-defined types. These types are `Fluent` which is defined as a set of all grounded fluent literals that occur in the specifications being processed, `Event` is the set of all event literals, and `Inst` which is the unique name of the institution being processed.

Event Declarations: InstAL allows us to specify zero or more event signatures, each of which describes the event's status (exogenous, institutional or violation), its (unique) name and the types of any parameters associated with the event in form of

$$event - type \texttt{ event } event - name(parameters);$$

Following Section 3.6.2 on page 59, event types include; `exogenous` which declares external (observable) events, `inst` for internal institutional events, `violation` for violation events, and `creation` for creation event. For example, some of our events declarations are

```
exogenous event prepare(Lecturer,Paper);
inst event iprepare(Lecturer,Paper);
```

As seen in these examples, we parametrise event signatures using the types described in the institution such that each event signature describes one or more event literals which are derived from the domains of the types of the event's parameters. The set of all event literals for a particular event type is described as the domain of that event. Where an event does not have parameters, the domain of the event declaration contains a single event with the same name as the declaration.

Fluent Declarations: Fluent declarations define institutional properties which may change over time. These are categorised into institutional (normative) fluents and domain specific fluents (see Section 3.6.2 on page 60). A fluent declaration consists of a fluent name and zero or more fluent parameters. The InstAL implementation we are using in this work called *pyinstal*, allows us to

define three types of fluents. These are *inertial fluents*, *noninertial fluents*, and *obligation fluents*.

Domain specific fluents are declared differently depending on the type. *inertial* fluents take the following form:

`fluent-type` $fluent - name(parameters)$;

For instance the following declarations:

```
fluent hasPaper(Head);
fluent setToTake(Student,Paper);
```

define fluents with names `hasPaper` with one parameter and `setToTake` with two parameters which range over the types `Head`, `Student` and `Paper` respectively. Each fluent declaration corresponds to a set of possible domain fluents which may be used in the institution, the full set of fluents described by a given fluent declaration is dependent on the types of the parameters of that fluent. For example if the type `Head` is defined by the values `hodPhysics`, `hodMaths`, and `hodChemistry` then the fluents `hasPaper(hodPhysics)`, `hasPaper(hodMaths)` and `hasPaper(hodChemistry)` are considered to be valid fluents in the institution. The set of all literals corresponding to a particular fluent is referred to as the domain of the fluent.

The declaration of noninertial fluents take the form:

`noninertial fluent` $fluent - name(parameter...)$;

This declares a fluent that is non-inertial. This is a fluent that is not automatically persistent between states but only come about as a result of the presence or absence of some specified fluents.

Obligations are treated as fluents and are declared as follows:

`obligation fluent obl`(e, d, v);

Similar to events, where a fluent has no parameters the domain of the fluent declaration will contain only a single literal corresponding to the fluent name.

In addition to the domain fluents declared in a specification the following institu-

tional fluents are implicitly defined. These are the permission fluent `perm(`*Event*`)` and power fluent `pow(`*Event*`)`

Rules Declarations: Each Inst*AL* specification may contain zero or more rules. Three types of rules which can be declared are;

(i) **Consequence rules** which describe when fluents change in response to the occurrence of events. A causal rule consists of:
   (a) *a trigger event* which denotes the event which (may) activate the rule.
   (b) *an operation* which indicates whether the rule `initiates` or `terminates` the fluents in the rule body.
   (c) *a set of fluents* which are initiated or terminated by the rule.
   (d) *a (possibly empty) condition* consisting of an expression describing fluents which must be true in order for the rule to have an effect.

   Fluent initiation rules are declared as:
   triggerEvent `initiates` *institutionalFluent* [*condition*];
   which corresponds to the definition of the set $\mathcal{C}^{\uparrow}$ in the underlying formal institutional model.
   Similarly, rules that terminate fluents are declared as:
   triggerEvent `terminates` *institutionalFluent* [*condition*];
   which also correspond to the definition of $\mathcal{C}_{\downarrow}$ in the formal institutional specification.
   For instance in our specification we have the following `initiates` rule;

```
iprepare(Lecturer,Paper) initiates perm(sendPaper(Lecturer,Head)),
       perm(isendPaper(Lecturer,Head)), pow(isendPaper(Lecturer,Head)),
       hasPaper(Head);
```

   This rules specifies that every time the institution recognises the occurrence of the event `iprepare(Lecturer,Paper)` in which the `Lecturer` prepares a `Paper`, the event initiates power and permissions for the lecturer to send paper to the head and for the event to be recognised by the institution as a valid event. It also initiates the fluent `hasPaper` which indicates the state of the `Head` with respect to the institution which persists from the next time instant except terminated by another event.

(ii) **Generation rules:** corresponding to the definition of the event generation relation $\mathcal{G}$ described in Section 3.6.2, the generation rules describe when

events may be generated. In InstAL, a generation rule description consists of a *triggerevent* which may activate the rule, a set of *generatedevents* which result from the occurrence of the trigger event specified in the rule, and a possibly empty *condition* describing fluents that are required to be true for the rule to be effective. This rule therefore takes the general form of:

triggerEvent `generates` *generatedEvents* [*condition*];

The generation function $\mathcal{G}$ in the formal model referenced maps a condition expression and a single event to a set of generated event literals. Hence for a given generation rule as described, the formal model is represented by the following components: The trigger event in the rule represents the event literal $\mathcal{E}$ in $\mathcal{G}$. The conditions of the rule correspond to fluent expressions $\mathcal{X}$ in the $\mathcal{G}$ relation for the given trigger event. The generated event literals described by the rule correspond to set of events in the range of the $\mathcal{G}$ relation for the given trigger event $\mathcal{E}$ and condition $\mathcal{X}$.

An example follows from our scenario specification thus;

```
prepare(Lecturer,Paper) generates iprepare(Lecturer,Paper);
```

In this example, the condition is empty, however the rule is still valid. This rule caused the `prepare` event to generate the `iprepare` event.

(iii) **Initial rules:** These rules correspond to the definition of the set of initial fluents Δ in our formal model of institutions. The rules describe the initial state of the institution whenever the institution is created. Therefore for any institution being modelled, every fluent literal in an expanded initially rule corresponds to a member of the set Δ. The declaration is achieved using the keyword `initially` followed by one or more fluent expressions i.e.;

`initially` *fluentExpression*(s) [*condition*];

An example from our specification follows:

```
initially
       perm(prepare(lecturer,paper)),
       pow(iprepare(lecturer,paper)),
       perm(iprepare(lecturer,paper));
```

which states all the fluents that would be true when the institution is created.

We have given brief overview of the features of InstAL as they would be used in our modelling. We further summarise the specific features we shall be using in Table 4.4.

InstAL Feature	**Declares...**	**Example**
`institution` *name*	declares name of the institution	`institution examPaper;`
`type` *identifier*	declares a type	`type Lecturer;`
`exogenous event` *event − name*(*type*$^+$)	a new physical world event with its parameter types	`exogenous event prepare(Lecturer,Paper);`
`inst event` *event − name*(*type*$^+$)	a new institutional event with its parameter types	`inst event iprepare(Lecturer,Paper);`
`violation event` *event − name*(*type*$^+$)	a new violation event	
`fluent` *fluent − name*(*type*$^+$)	a new institutional fact	`fluent setToTake(Student,Paper);`
`noninertial fluent` *fluent − name*(*type*$^+$)	a non-inertial fluent (fluent which does not persist between states automatically)	
event − name `generates` *institutional − event*$^+$[*condition*]	the generation of a new institutional event optional condition	`prepare(Lecturer,Paper) generates iprepare(Lecturer,Paper);`
event − name `initiates` *institutional − fluent*$^+$[*condition*]	adds a new pair to the consequence (addition) relation, with domain event (physical or institutional) and range fluent.	`itakes(Student,Paper) initiates hasTaken(Student,Paper);`
event − name `terminates` *institutional − fluent*$^+$[*condition*]	adds a new pair to the consequence (deletion) relation, with domain event (physical or institutional) and range fluent.	`iprepare(Lecturer,Paper) terminates perm(prepare(Lecturer,Paper)), perm(iprepare(Lecturer,Paper)), pow(iprepare(Lecturer,Paper));`
noninertial − fluent − name `when` [*condition*]	adds the conditions in which a noninertial fluent should be true in a given state	
`perm`(*event*)	permitted events, typically the subject of `initiates` or `terminates` rules	`perm(sendPaper(Lecturer,Head))`
`pow`(*event*)	empowered events, also typically the subject of `initiates` or `terminates` rules	`pow(isendPaper(Lecturer,Head))`
`obl`(*event*, *event*, *event*)	obligations	

Table 4.4: Summary of the relevant InstAL features to this work

4.7 Summary of Chapter

In this chapter we presented the Answer Set Programming (ASP) paradigm along with the semantics of *AnsProlog*. ASP is a logic programming language which allows for non-monotonic reasoning. Its handling of negation as *negation as failure* and *constraint-based negation* makes reasoning about incomplete information possible. We compared ASP with other logic reasoning paradigms including satisfiability solving (SAT) and Prolog. We noted the advantage that *AnsProlog* has over these other computational reasoning approaches which is the fact that it allows for the specification of both the problem as well as the query for the answer set as a single executable program which results in a more straight forward verification and validation task. The syntax and semantics of ASP were also presented. Furthermore we presented the translation of our institutional framework into ASP and introduced the actions language InstAL. InstAL is a language which was developed for the purpose of specification of institutions. It abstracts from ASP by allowing institutions to be specified in a more human readable format.

Chapter 5

CVF: The Computational Verification Framework for Security Requirements

5.1 Introduction

We pointed out in Chapter 2 the lack of a computational means for the verification of elicited security requirements associated with the approaches reviewed. The approaches are not centred on the human factor aspect of information security in which human interactions (considered as social in this context) constitute possible sources of security breaches. Also, since these approaches are driven through manual processes it is very difficult for the validation and analysis of the elicited requirements. In order to complement these approaches, we are providing a computational approach to the verification and validation of elicited security requirements. To do this, we are using the institutional framework which we have presented in chapters 3 and 4.

In this Chapter we present our computational verification framework (CVF) for the analysis of security requirements and properties.

5.2 CVF: Computational Verification Framework

Our framework CVF is based on an institutional framework which provides a means for us to capture and reason about what is *correct* and *incorrect*. In the context of organisational information security management which is the focus of

our work, *correct* behaviour would be the behaviour that preserves the security of information assets, or behaviour that complies with security rules and regulations while an *incorrect* behaviour will be such a behaviour that causes a violation of the security of the information assets whether maliciously or not. The overview of our framework is presented in Figure 5-1. We view our methodology as consisting

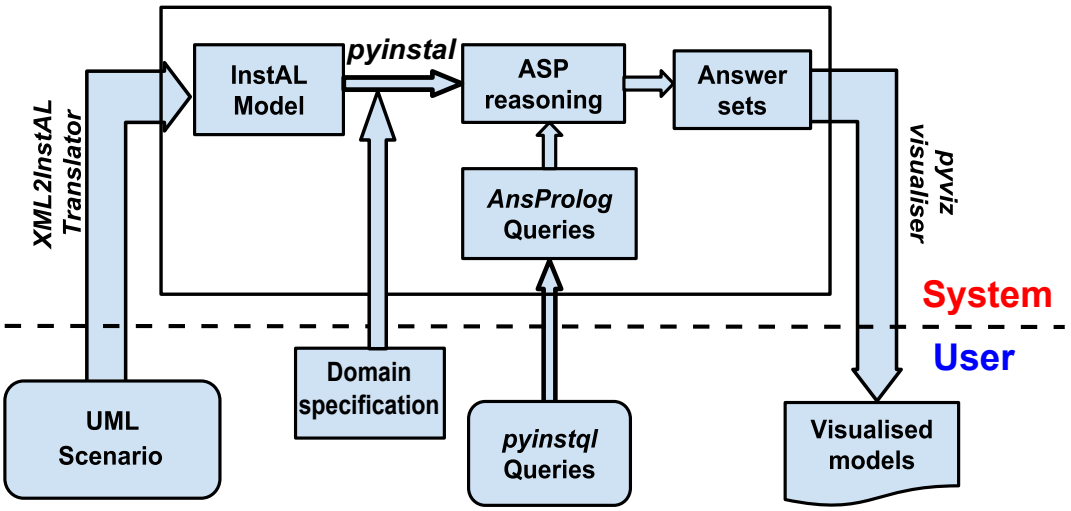


Figure 5-1: An Overview of the Computational Verification Framework

of two parts: the user part and the system part.

Our framework starts with a UML design of a scenario. By designing scenarios in UML we take advantage of the standard symbolic notations of UML. UML tools are also commonly used and understood in practice. Therefore it comes readily as the choice of tool for our scenarios. Although there is no unified formal semantics for UML activity dagrams, we have developed formal semantics for our translation based on some related earlier works on formalisation of semantics of UML activity diagrams (van der Aalst and van Hee, 2004; Bouabana-Tebibel and Belmesk, 2007; Xu et al., 2008) which is presented in Section 6.2.1. The XML representation of the UML scenario is passed to the XML2InstAL tool which we developed for the extraction of relevant information needed for the InstAL specification of the model. The detailed description of the XML2InstAL tool is presented in Chapter 6 where we present the tools developed in the course of this work.

Next, the InstAL model generated from the XML representation of the scenario

is then translated to the computational logic program model in *AnsProlog* as described in Chapter 4 on page 63 using the existing *pyinstal* tool. The domain specification which specifies variables for the purpose of grounding is also supplied to *pyisntal* which produces a grounded logic program. InstAL is already an abstraction from ASP suitable for modelling institutions, hence it makes more sense to us to translate our XML representation to InstAL instead of ASP directly.

Verification is carried out by specifying queries just as one would query a database for information. Our queries are written in *pyinstql*, a translation tool which takes queries specified in a semi-high-level form and translates it to an equivalent *AnsProlog* query specification. The *pyinstql* tool is described in Chapter 6.

Through the use of the existing ASP system *clingo*, the logic program produced by *pyinstal* is *solved* along with the query specification to produce answer sets. However the answer sets produced are in text form and usually complicated to understand. We therefore developed the *pyviz* tool which visualises the answer set in a graphical and easy to read form. We also present this tool and illustrations of its usage in Chapter 6.

Having given an overview of our methodology, in the next section we present a case study through which we walk through the process in a general sense. Using misuse case analysis for the initial elicitation of security requirements, our solution provides a means for a rigorous test of the security requirements elicited. The combination of our approach with already established misuse case approach provides a tool that would be more useful for effectively determining a system's security requirements at design time.

5.3 Case Study: Patient Referral Management

The publicly available iTrust Medical Records System documentation (Williams et al., 2011) provides ample choice of use-cases for the purpose of illustration. After consideration of the use-cases we chose one that we found most suitable for illustrating the institutional framework approach. The criteria for the choice of use-case is that the scenario should consist of more than one actor and a well defined process in which the actors interact in order to achieve a certain

goal or sub-goal of the system. This is necessary in order for us to be able to express the interactions as events, while the actors are taken as agents, with the aim of generating the traces of events which could be investigated for some desired security properties. The use case is described in Table 5.1. The use case descriptions as presented here are based on popular templates proposed in the literature (Cockburn, 2001; Kroll and Kruchten, 2003; Kulak and Guiney, 2004). Templates guide authors in writing a clear and simple flow of events when describing use cases. Also templates prompt authors to consider adding certain information about a use case that might otherwise be overlooked. The use of templates is therefore encouraged in practice since they result in higher quality use cases in comparison to use cases developed without the use of templates (Achour et al., 1999; El-Attar, 2012b).

UC33: Manage Patient Referrals Use Case

33.1 Preconditions:
A patient and two HCPs are registered users of the iTrust Medical Records system (UC2). The iTrust user has been authenticated in the iTrust Medical Records system.

33.2 Main Flow:
A sending HCP refers the patient to another receiving HCP [S1]. A receiving HCP views a list of received referrals [S2]. A sending HCP views a list of previously sent patient referrals [S3]. A patient views the details of his/her referrals [S4]. A sending HCP edits a previously sent patient referral [S5]. A sending HCP cancels a previously sent patient referral [S6]. All events are logged.

33.3 Sub-flows:

- *[S1]* An HCP chooses to refer a patient to another receiving HCP through the referral feature on a patient's office visit page. The sending HCP must select a receiving HCP by either entering the HCP's MID and confirming the selection, or by searching for the HCP by name. The sending HCP is also presented with a text box to include notes about the referral. The sending HCP then chooses a priority from 1-3 (1 is most important, 3 is least important) for the referral. The HCP may send the referral, cancel the referral [E1], or edit the referral [E2]. Upon sending a referral, the patient, sending HCP, and receiving HCP receive a message summarizing the newly created referral information (sending HCP name & speciality, receiving HCP name & speciality, patient name, referral notes, and referral creation time-stamp); additionally, the sending and receiving HCP messages include the referral priority.
- *[S2]* An HCP chooses to view received referrals. The receiving HCP is presented with a list of referrals sorted by priority (from most important to least important). The receiving HCP then selects a referral to view details and is presented with the name and speciality of the sending HCP, the patient's name, the referral notes, the referral priority, the office visit date with a link to the office visit, and the time the referral was created.
- *[S3]* A sending HCP views a list of previously sent patient referrals. The HCP may sort the list of referrals by patient name, receiving HCP name, time generated, and/or priority. The HCP chooses a specific referral from the list to view complete details about the referral: patient name, receiving HCP name and speciality, time generated, priority, office visit date, and notes.
- *[S4]* A patient views a list of his/her referrals. The patient may sort the list of referrals by receiving HCP name, time generated, and/or priority. The patient chooses a specific referral from the list to view complete details about the referral: sending HCP name and speciality, receiving HCP name and speciality, time generated, priority, office visit date, and notes. The patient is also provided with the option to send a message to the receiving HCP to request that an appointment be scheduled.
- *[S5]* A sending HCP edits a previously created patient referral as long as the referral has not been viewed by the receiving HCP. The sending HCP may edit the priority of the referral and/or the referral notes. The sending HCP then chooses to save the edits, cancel the edits, or re-enter the data [E2].
- *[S6]* A sending HCP cancels a previously sent patient referral by visiting the office visit page, viewing the details of a previously sent patient referral [S3], and choosing cancel. The HCP is asked to confirm the decision to cancel the referral. The patient and receiving HCP receive a message indicating that the referral was cancelled.

33.4 Alternative Flows:

- *[E1]* The receiving HCP chosen is not the desired HCP. The sending HCP does not confirm the selection and is prompted to try again.
- *[E2]* The patient, receiving HCP, referral notes, and/or referral priority are invalid, and the HCP is prompted to enter this information again.

Table 5.1: Managing Patients Referrals Use Case (Williams et al., 2011)

5.3.1 The "Making Referrals" Scenario

This scenario consist of two healthcare professionals (designated here as **sHCP** and **rHCP**) and a patient. The process involves the sHCP referring a patient to rHCP. The sHCP initiates the process by login to the system, prepares the referral document, and sends the referral document to rHCP, the patient and him/herself. While the sHCP and rHCP receive similar copies of the referral document, the patient receives a slightly different version.

This scenario was first analysed for possible security threats using the misuse case approach illustrated in Figure 5-2. The classes of threats identified include *impersonation, escalation of privileges, and information tampering.* We consider these attacks as being the kind of attacks that could be carried out by insider attackers (escalation of privileges) and external attackers (impersonation and tampering). The original misuse-case in Sindre and Opdahl (2005) was modified to include the security goals under threat at each stage of the process. This is a simplified version of the one proposed in Okubo et al. (2009). The reasoning behind the modification is that security threats are primarily aimed at breaching security goals with respect to confidentiality, integrity, availability, and accountability. Inclusion of these in the use/misuse case diagram would make it easier to see the type of threat/attack an attacker would consider. It also helps in determining the appropriate countermeasure to apply for the attacks or threats. The security goals relevant in each scenario is drawn from the analysis of the most relevant assets associated with the scenario. Hence the approach is consistent with Okubo et al. (2009).

The detail interactions for this use/misuse case description is illustrated in Figure 5-3. This is the mal-activity diagram, based on Fernandez et al. (2006); Okubo et al. (2011), showing the users, sequence of events, the attackers, attack points, threats, and countermeasures. The mal-activity diagram describes the expected order and sequence of events which include the user initiated events and the attacker/misuser events. It also shows the countermeasures to the identified security threats and the points at which the countermeasures are expected to be applied in order to mitigate appropriately the threats. At this point, the misuse case security requirements elicitation and analysis phase ends. Usually the process may be repeated, depending on whether the countermeasures introduce new threats to the system or not. However, i) the level of refinement is totally based on the expertise and experience of the security analyst, and ii) the misuse case approach lacks the means of verifying and validating the behaviour

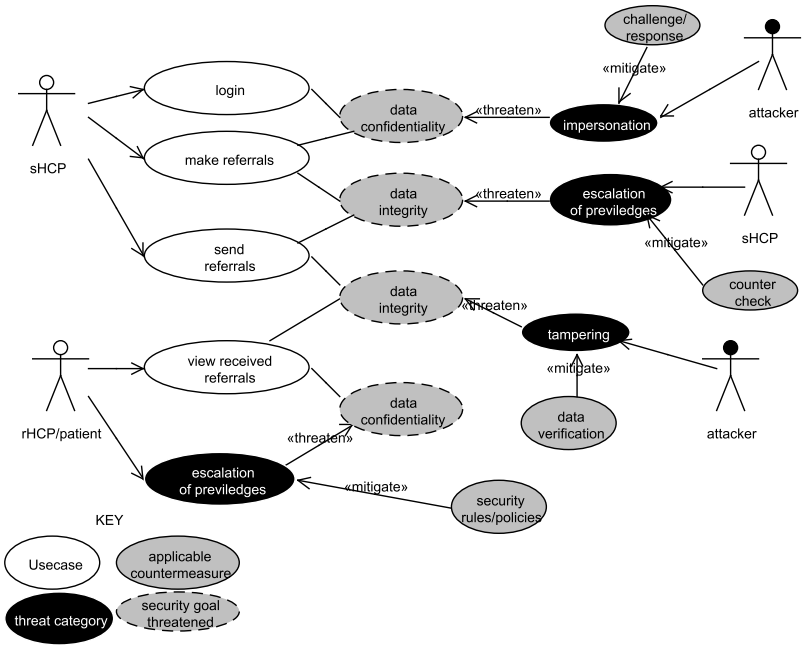


Figure 5-2: The *Make Referrals* Misuse Case Diagram

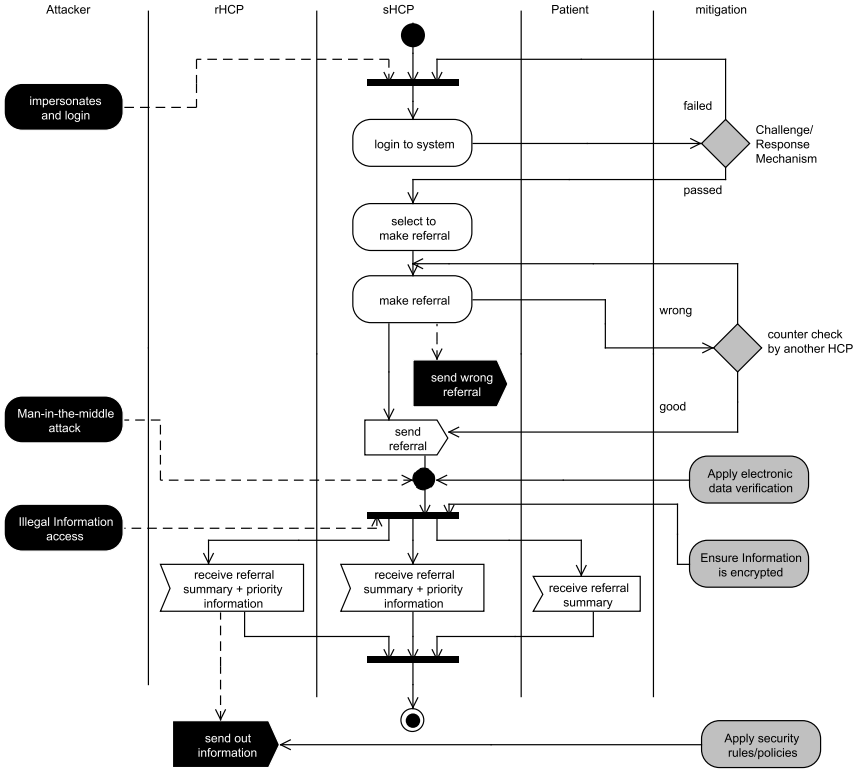


Figure 5-3: Mal-activity Diagram for *Make Referral* Scenario

```
1  institution patientRef;
2  % Type declaration
3  type HCP;
4  type Patient;
5  type System;
6  type Checker;
```

Figure 5-4: Type declaration for the *Make Referral* scenario.

of the system when the identified countermeasures are applied. We therefore introduce the event-based institutional framework, implemented using the Institutional Action Language (InstAL) . This provides the needed reasoning mechanism for the purpose of verification and validation of the misuse case analysis.

5.3.2 Misuse Case Implementation using InstAL

In carrying out the verification and validation of the misuse case description in Figure 5-2, we made use of the mal-activity diagram in Figure 5-3. Since we are doing a static analysis here, we simply assume that attacker events could always happen at any time. We are therefore concerned with the users' events and the countermeasures. The mal-activity diagram provides the necessary sequence of activities which would be represented as events in the institution specification. The complete implementation in InstAL consist of a set of declarations which we shall be describing here.

The InstAL institution specification begins with the declaration of the institution name and types (See Figure 5-4). The types here represent the various actors (agents) that would be interacting in the system. Notice that there is no representation of the attacker or misuser here. This is because the activities of the misuser/attacker is not explicitly modelled in this system.

Next is the declaration of events (Figure 5-5). This consist of four kinds of event declarations;

- *Exogenous events* consist of all observable real world events as described in Section 3.6.2. These events would consequently generate institutional events and cause changes to the institutional state. The *creation event* is responsible for the creation of the institution.
- *Institution events* consist of the various events that would be generated in the institution framework as a result of the occurrence of exogenous events. These

```
% Exogenous events
exogenous event login(HCP);
exogenous event loginChal(System,HCP);
exogenous event selectAction(HCP);
exogenous event makeRef(HCP);
exogenous event checkRef(Checker);
exogenous event signRef(System);
exogenous event encryptRef(System);
exogenous event sendRef(HCP);
exogenous event logindl;
exogenous event checkRefdl;
exogenous event receiveRefwP(HCP);
exogenous event enforcePolicy(System);

% creation event
create event create_patientRef;

% Institutional events
inst event ilogin(HCP);
inst event iloginChal(System,HCP);
inst event iselectAction(HCP);
inst event imakeRef(HCP);
inst event icheckRef(Checker);
inst event isignRef(System);
inst event iencryptRef(System);
inst event isendRef(HCP);
inst event ilogindl;
inst event icheckRefdl;
inst event ireceiveRefwP(HCP);
inst event iapplyPolicy(HCP);

% violation events
violation event loginChalCompromised;
violation event refUnchecked;
```

Figure 5-5: Events declaration for the *Make Referral* scenario.

events may initiate new facts in the institution, thereby resulting in a change in the institutional state.

- *Violation events* declare events that would occur whenever there is a violation in the system, such as failure of obligations.

Following the events declaration is the declaration of *fluents* (Figure 5-6). Fluents denote facts that may be present in the system state that can be added or deleted as a result of the occurrence of events in the system. *Non-inertial* fluents e.g. `loginAttacked`

```
41 % fluents
42 fluent chalSuccess;
43 fluent chalFail;
44 fluent loginCompromised;
45 fluent goodRef;
46 fluent badRef;
47 fluent hasRef(Patient);
48 fluent hasRefwP(HCP);
49 fluent misdCheck;
50 fluent refSigned;
51 fluent refEncrypted;
52 fluent policyApplied(HCP);
```

Figure 5-6: Fluents declaration for the *Make Referral* use-case.

in line 53 are used to capture boolean relationships over several fluents.

We now describe the generation and consequence relations of the model in three phases for simplicity and ease of understanding. The three phases are *login, create referral, send referral.*

The *login* Phase

The sequence of events starts off with the login event. The *login* phase specifies the occurrence of the login event and the rules that apply in order to counter the expected attack at the login phase. From Figure 5-3, it is assumed that an external attacker can impersonate after possibly acquiring a user's login information. We are not concerned about how the attacker acquires the login information here. Since this attack is assumed to happen at any time, we therefore specify the countermeasure, which in this case is a challenge-response event initiated by the system the moment there is an initial login event. We are not concerned about the detailed implementation of this countermeasure (which could be a series of events), but rather we treat it as a single event which we expect to happen at and within some time interval. The challenge-response event `loginChal(System,HCP)` is specified as an obligation (Figure 5-7, line 60) which must happen before a deadline event happens, else it triggers a violation event `loginChalCompromised`.

The challenge-response event is expected to be triggered just once and by the deadline, the countermeasure is either successful or fail and the institution state is set accordingly. The next exogenous event in the system's sequence of events can only take place when

```
53 % trigger challenge-response upon initial login
54 login(HCP) generates ilogin(HCP);
55 logindl generates ilogindl;
56 loginChal(System,HCP) generates iloginChal(System,HCP);
57 ilogin(HCP) initiates perm(loginChal(System,HCP)),
58   perm(iloginChal(System,HCP)), pow(iloginChal(System,HCP)),
59 %%this should happen before some deadline by which the next event
       would be triggered
60   obl(loginChal(System,HCP),logindl,loginChalCompromised);
61
62 %%challenge response triggered once
63 iloginChal(System,HCP) terminates perm(loginChal(System,HCP)),
64   perm(iloginChal(System,HCP)), pow(iloginChal(System,HCP));
65
66 %% challenge-response sets a state of success or failure
67 iloginChal(System,HCP) initiates chalSuccess;
68 iloginChal(System,HCP) initiates chalFail;
69 iselectAction(HCP) terminates chalFail;
70
71 %% institutional state is set to indicate an attack at login when
       challenge fails
72 always loginAttacked when chalFail;
73
74 %% the institutional state is set to indicate a compromise when
       obligation fails
75 loginChalCompromised initiates loginCompromised;
76
77 %%challenge response is triggered again whenever the obligation
       fails
78 loginChalCompromised initiates perm(loginChal(System,HCP)),
79   perm(iloginChal(System,HCP)), pow(iloginChal(System,HCP)) if
        loginCompromised;
```

Figure 5-7: Generation and Consequence relations for the *login* phase of *Make Referral* model.

there is no violation at this stage.

Create Referral Phase

In this phase, we specify the rules and the conditions for the creation of the referral. The security threats to the system which falls within this phase include escalation of privileges (an authorized user misusing his privileges), man-in-the-middle attack, and illegal access to information. Since it is equally assumed that any or all of these attacks can always happen and at any time, the countermeasures are also specified in this phase.

```
selectAction(HCP) generates iselectAction(HCP);
ilogindl initiates perm(selectAction(HCP)),pow(iselectAction(HCP)),
  perm(iselectAction(HCP)) if chalSuccess, not loginCompromised;

%HCP can then create the patient referral information
makeRef(HCP) generates imakeRef(HCP);
iselectAction(HCP) initiates perm(makeRef(HCP)),
  pow(imakeRef(HCP)), perm(imakeRef(HCP));

%% action selected once
iselectAction(HCP) terminates perm(selectAction(HCP)),
  pow(iselectAction(HCP)),perm(iselectAction(HCP));

%the event to proof the referral is trigered
checkRef(Checker) generates icheckRef(Checker);
checkRefdl generates icheckRefdl;
imakeRef(HCP) initiates perm(checkRef(Checker)),
  perm(icheckRef(Checker)), pow(icheckRef(Checker)),
%% and expected to occur before some deadline
  obl(checkRef(Checker),checkRefdl,refUnchecked);

%%terminate the ability to create the referral
imakeRef(HCP) terminates perm(makeRef(HCP)),pow(imakeRef(HCP)),
  perm(imakeRef(HCP));% if patientRefCreated(HCP);

%%proof can either set the state to good or bad
icheckRef(Checker) initiates goodRef;
icheckRef(Checker) initiates badRef;
isignRef(System) terminates badRef;

%%obligation failure sets institutional fluent
refUnchecked initiates misdCheck;

%% sign the referral once if checked and good
signRef(System) generates isignRef(System);
icheckRefdl initiates perm(signRef(System)), pow(isignRef(System)),
  perm(isignRef(System)) if goodRef, not misdCheck;
isignRef(System) terminates perm(signRef(System)),
  pow(isignRef(System)), perm(isignRef(System));

%% remake the referral if bad
icheckRefdl initiates perm(makeRef(HCP)),
  pow(imakeRef(HCP)), perm(imakeRef(HCP)) if badRef;

%%encrypt referral once after signing
encryptRef(System) generates iencryptRef(System);
isignRef(System) initiates refSigned, perm(encryptRef(System)),
  pow(iencryptRef(System)), perm(iencryptRef(System));
iencryptRef(System) terminates perm(encryptRef(System)),
  pow(iencryptRef(System)), perm(iencryptRef(System));
```

Figure 5-8: Generation and Consequence relations for the *create referral* phase of *Make Referral* model.

The escalation of privileges threat is expected to be mitigated by initiating a check on the created patient referral for correctness. This is specified as a `checkRef(Checker)` event which must happen within a certain time span, else a violation event `refUnchecked` is triggered (Figure 5-8, line 99). The completion of this countermeasure event will set the institution state at this point to either `goodRef` indicating the referral was good and hence satisfying the condition for the next event to happen, or `badRef`, indicating that the referral was bad, in which case the referral would have to be corrected (Figure 5-8, line 107).

The man-in-the-middle attack is mitigated by applying some form of digital signature on the prepared and proofed referral such that any form of tampering could be detected. This is specified in the model as an event `signRef(System)` which will be permitted to happen only if the referral has been checked and it is good (Figure 5-8, lines 11-118). It is also expected that the signature would be applied only once, hence the power and permission for this event is terminated once it has happened.

Encryption is taken as the countermeasure for illegal access of information. This is specified in the model as an event `encryptRef(System)` which would happen after the signature event has happened.

The *Send Referral* Phase

This is the final phase of the process we are considering. In this phase, we still see escalation of privileges as a possible threat to the confidentiality of the patient's vital information which could be contained in the referral information. The receiving health care professional (rHCP) to whom the patient has been referred could send out the information to a third party. There is therefore the need to institute appropriate rules or policies that would prevent the rHCP from carrying out such activities. This is specified in the model as an event whose permission would be initiated when the rHCP receives the referral information. The permission remains through the life time of the institution.

```
130 %%send referral once after encryption
131 sendRef(HCP) generates isendRef(HCP);
132 receiveRefwP(HCP) generates ireceiveRefwP(HCP);
133 enforcePolicy(System) generates iapplyPolicy(HCP) if
        hasRefwP(HCP);
134 iencryptRef(System) initiates refEncrypted, perm(sendRef(HCP)),
135   pow(isendRef(HCP)), perm(isendRef(HCP));
136 isendRef(HCP) terminates perm(sendRef(HCP)),
137   pow(isendRef(HCP)), perm(isendRef(HCP));
138
139 isendRef(HCP) initiates
        perm(receiveRefwP(HCP)),pow(ireceiveRefwP(HCP)),
140   perm(ireceiveRefwP(HCP)), hasRef(Patient),hasRefwP(HCP);
141
142 %%set states to indicate the presence of referrals
143 isendRef(HCP) initiates hasRef(Patient),hasRefwP(HCP),
144   perm(iapplyPolicy(HCP)), pow(iapplyPolicy(HCP));
145 %%security policies should be applied by the HCP upon receiving
        referrals
146 iapplyPolicy(HCP) initiates policyApplied(HCP) if hasRefwP(rHCP);
```

Figure 5-9: Generation and Consequence relations for the *send referral* phase of *Make Referral* model.

Initialisation

Finally, we declare the initial state of the institution. This represent the events that are permitted to happen from the start of the institution. These are expressed in InstAL as follows;

```
initially
perm(login(sHCP)), pow(ilogin(sHCP)), perm(ilogin(sHCP)),
perm(logindl),pow(ilogindl),perm(ilogindl),
perm(checkRefdl),pow(icheckRefdl),perm(icheckRefdl),
perm(enforcePolicy(sys)),perm(iapplyPolicy(rHCP)),
pow(iapplyPolicy(rHCP));
```

Some of these initial states may be terminated in the course of the evolution of the institution. For example `iloginChal(System,HCP) terminates perm(loginChal(System,`

`HCP)), perm(iloginChal(System,HCP)), pow(iloginChal(System,HCP))` (lines 63-64 Figure 5-7) deletes the corresponding fluents from the institutional state at the next time instant after the event `loginChal(System, HCP)` has occurred.

Domain Specification

So far, we have given the complete specification of our *Make Referral* scenario. However, there is also a need to specify the domain information which would be used for grounding (described in Section 4.2 on page 63) some aspects of the institution when translating the InstAL institution to $AnsProlog$. This is specified, providing arguments for each of the declared types in the institutional specification as presented in Figure 5-10.

```
1 HCP: sHCP rHCP
2 Patient: pat01
3 System: sys
4 Checker: cHCP
```

Figure 5-10: Domain Specification for the *Make Referral* model.

5.4 Results: Trace and Query

In Section 4.3 we introduced the concept of a trace program which allows us to model (as answer sets) of one or all possible traces of the institution when combined with an institution program $\Pi_{\mathcal{I}}^{base(n)}$.

The result (answer set, in the language of ASP) provides a rich database of information in form of ordered traces of events. The use of queries, as the experience from databases would suggest, provides a means of examining the traces for any system behaviours that might be of interest. The properties being investigated could be expressed as facts and/or rules. The results can then be interpreted and decisions or actions taken appropriately.

Query formulation and trace construction are intimately tied up. However, before traces can be generated, the program must be grounded, which means being explicit about the meaning of variables and time instants, defining precisely how

many there are, which in turn determines the length of the trace. For time instants $t_i : 0 \leq i \leq n$, we define the following three rules: `instant(`t_i`)`, `next(`t_i`,` t_{i+1}`)` and `final(`t_n`)`, denoting each ground instant of time, relative order and final time instant, respectively. The grounding variables are provided in a domain file which is passed to the ASP translator along with the instAL description file.

The general trace program generates the answer sets containing all possible combinations of n exogenous events, but by the addition of constraints, the answer sets can be limited to those containing desired traces. This is expressed in the form of a query specification. For example, Figure 5-11 shows how the answer set can be constrained to the consequences of specific observed events[1].

```
observed(login(sHCP),1).
observed(loginChal(sys,sHCP),2).
observed(logindl,3).
observed(selectAction(sHCP),4).
observed(makeRef(sHCP),5).
observed(checkRef(cHCP),6).
observed(checkRefdl,7).
observed(signRef(sys),8).
observed(encryptRef(sys),9).
observed(sendRef(sHCP),10).
observed(receiveRefwP(rHCP),11).

#hide.
#show occurred(E,I).
```

Figure 5-11: Verifying the expected sequence of observed events for the *Make Referral* model.

The result, as listed in Figure 5-12 on the following page verifies the correctness of the model, in that first of all, events occurred in the expected sequence, subject to the flag `#show occurred(`E`,` I`)`, which is read as "show event E occurring at instant I".

[1]Real world events are tagged `observed` in traces, while institutional ones are tagged `occurred`.

```
Answer: 1
occurred(create_patientRef,i00) occurred(login(sHCP),i01)
occurred(loginChal(sys,sHCP),i02) occurred(logindl,i03)

occurred(ilogin(sHCP),i01)
occurred(iloginChal(sys,sHCP),i02) occurred(ilogindl,i03)
SATISFIABLE
```

Figure 5-12: Output of the model verification query for the *Make Referral* model.

This way we are able to test the correctness of the model in the sense that observed events occur at the right time instants. With this, the model can be used to investigate any behaviour of interest which in this case could be verification of occurrence of violation events, effects of such violations, and perhaps the security state of the system after or before the occurrence of certain events.

In this scenario, the criteria for a successful and secure process would be that the countermeasure events were successfully observed at the proper time instants. We can therefore examine the states of the institution at the final instant to see if those conditions or countermeasures actually holds. Such a query is written as a rule:

```
success:- holdsat(chalSuccess,F),
  holdsat(goodRef,F),
  holdsat(refSigned,F),
  holdsat(refEncrypted,F),
  holdsat(policyApplied(rHCP),F),

  not holdsat(misdCheck,F),
  not holdsat(chalSkipped,F),
  not holdsat(loginAttacked,F),
  not holdsat(policyViolated(rHCP),F),

  final(F).
```

These states should be reachable and remain true up to the final instant

States that are not expected to be true at the finale instant

defines the final instant of institutional time

It is possible to see traces of events that happened before a particular state of the system. For instance, the following query shows traces before the state `loginCompromised`.

```
happened(E,I0) :-
  holdsat(loginCompromised, i03),
  occurred(E,I0), before(I0,i03),
  instant(I0).
```

This ensures that events at instant **I0** occur before event at instant **i03**

This query provides the following result:

```
happened(createpatientRef,i00)
happened(ilogin(sHCP),i01)
happened(login(sHCP),i01)
happened(ilogindl,i02)
happened(vloginChalCompromised,i02)
happened(logindl,i02)
```

These events occured which led to the state of the system at instant **i03**

The security designer can also know the consequences of a violation event occurring. Figures 5-13 and 5-14 show the query specification and the resulting traces.

```
canHappen(E,I0):-
  occurred(viol(checkRef(cHCP)),I),
  occurred(E,I0), after(I0,I),
  instant(I),event(E).
```

This ensures the ordering of the events so that events at instant **I0** occur after the event at **I** has occurred.

Figure 5-13: Consequences of violation - Query.

```
Answer: 1
canHappen(viol(sendRef(sHCP)),i10)
canHappen(sendRef(sHCP),i10)
canHappen(viol(receiveRefwP(rHCP)),i11)
canHappen(receiveRefwP(rHCP),i11)
canHappen(enforcePolicy(sys),i12)
canHappen(viol(encryptRef(sys)),i09)
canHappen(encryptRef(sys),i09)
canHappen(viol(signRef(sys)),i08)
canHappen(checkRefdl,i07)
SATISFIABLE
```

These are the events that would occur as a result of the violation.

Figure 5-14: Consequences of violation - Result.

5.5 Generalization of case study

So far, we have used our case study to illustrate how validation and verification of security requirements can be effectively carried out at design time. We give a few illustrative examples here to show from where a designer might embark on a more comprehensive analysis. In Chapter 7 we demonstrate how our approach

can be used for analysing confidentiality and integrity requirements. Although a lot of work has been done on using misuse case approach to eliciting security requirements (Sindre and Opdahl, 2000; Sindre, 2007; Braz et al., 2008; Okubo et al., 2009; Fernandez et al., 2006), none of these has provided a computational means of verifying the security requirements elicited. This problem also applies to other approaches for security requirements elicitation. Since our idea captures the system specification with security requirements, it enables both system and security designers to capture the system-to-be in terms of actors or roles, events, and rules through the use of permissions, powers, and obligations.

5.5.1 Usability/Scalability

ASP is conceived as a tool for working with problems in the NP-complete domain and while nothing can be done to escape this fact, as is often the case, the practical situation of interest can often be sufficiently tightly specified to make the task computationally tractable. ASP has therefore been successfully applied in practical application domains. Some of these applications include decision support systems (Nogueira et al., 2001; Beierle et al., 2005), combinatorial search problems involving substantial amount of data such as planning (Gebser et al., 2012; Tu et al., 2011; Lifschitz, 2002), bio-informatics (Bodenreider et al., 2008; Erdem and Yeniterzi, 2009; Erdem et al., 2011), security analysis (Delgrande et al., 2009), product configuration (Soininen and Niemelä, 1999; Tiihonen et al., 2003), diagnosis (Eiter et al., 1999; Balduccini and Gelfond, 2003; Eiter et al., 2009) and multi-agent systems (De Vos et al., 2006; Son et al., 2009; Baral et al., 2010; Sakama, 2011; Pontelli et al., 2012) to mention a few. In addition to these evidences found in the literature, we summarise some important points from Section 4.3 to explain some of the issues that border on the scalability of our approach.

i.) **Size of an answer set**

One factor affecting the size of an answer set is the number of observable (exogenous) events. The general trace program $\Pi_{\mathcal{I}}^{all(n)}$ generates the answer sets containing all possible combinations of n exogenous events. The answer set generated can be quite large since it also includes all fluents and facts

provided. However, the addition of constraints as part of the answer set querying limits the size of the answer set to those containing only desired traces. For example, in the query presented in Figure 5-11 on page 122, `#hide` and `#show occurred(E,I)` ensure that only occurrences of exogenous events are contained in the resulting answer set as seen in Figure 5-12.

ii.) **Number of answer sets**

The number of answer sets generated is determined by the permutation of the number of exogenous events up to the time steps x specified for running the model. For n observable events, the required time steps is $x = n+1$ and the number of answer sets will be n^x. If the model is run for a time step less than x it would result in an incomplete answer set because some observable events would have been ignored. Running the model for time steps greater than x on the other hand would result in repetition of the process thereby generating more answer sets and longer traces than necessary.

Another way to curtail the explosion in the number of answer sets is by constraining the order of events. Even if the right time-steps are specified and the order of events is not constrained, it will still lead to a very large number of answer sets. This is can be limited by specifying the order of events in the answer set query. For example, with reference to our foregoing case study, we constrain the order of events by specifying each event as `observed(`*event*`,`*instant*`)` as follows;

```
observed(login(sHCP), 0).
observed(selectAction(sHCP), 1).
observed(makeRef(sHCP), 2).
observed(sendRef(sHCP), 3).
```

5.5.2 Completeness

The completeness of an answer set is based on the fact that given a model produced by an ordered trace of events in an institution, the set of ASP atoms corresponding to the events and fluents in the model are all supported by the rules in the translation. The correctness of the translation is established by the fact that given an institution and its corresponding translation in ASP, each atom

contained in an answer set of the ASP translation is supported by a corresponding fluent status or event occurrence in the model of the formal institution for that ordered trace (Cliffe, 2007). The correctness of the model can be verified by running a query which constrains the traces to specific observed events occurring in a specifically expected order and the presence or absence of certain fluents in the system state. When the result returns SATISFIABLE, it means that an answer set of the model satisfies the query. With this the model can be used to investigate any behaviour of interest which could be verification of occurrence of violation events, effects of such violations, and perhaps the security state of the system after or before the occurrence of certain events. Using few examples, we illustrate the verification of some properties in Section 5.4.

5.5.3 Validation of the framework

Our framework presents a computational means for reasoning about security requirements with focus on human behaviours. In real world security requirements which are aimed at regulating human behaviours with respect to preserving the security of information systems are presented as policies or rules. Our approach enables us to model scenarios in which the regulations are to be applied (that is the business process, for instance) along with the rules specified. The rules determine the *permissions*, *empowerments* and the *obligations* declared in our model. We treat confidentiality (secrecy) requirement issues as *permission* issues while integrity requirements are treated as *obligations*. The traces produced by the model are tantamount to a set of all possible interactions between the various participants with respect to the rules specified. The security analyst can therefore computationally investigate certain security properties. We note the following limitations;

- We have limited our focus to confidentiality and integrity properties, hence the approach cannot necessarily be readily applicable to other classes of security properties such as availability and accountability.

- The predictions we make are based on what the effect of a breach could be on the system rather than exploration of the possible security breaches. Also, if there is a security breach, we can predict the sequence of events

that could have possibly caused such a breach. In the real world, such information provides the security analyst with forehand knowledge of how the security requirements would perform with respect to a proposed system design.

- The reliability of the predictions depends on the correctness of the model.

In our models, we assume that a system is vulnerable to attack at any point in a process. By this assumption we mean that an attacker event can occur at any time in the life time of a process. This is a valid assumption in the real world because information security begins and ends with the participants in a system and the people that interact with the system externally either intentionally or otherwise (Whitman and Mattord, 2012, pp. 32) since their behaviours may or may not constitute a security risk and hence breach of the system security requirements. As a result, we did not focus on modelling the attacker events which can be many. We model a process as a normal work flow and then explore the possible states of the system whenever an attack occurs.

The model we present here has a practical value applicable to security evaluation and risk management for information systems. The ability of the model to present a rich database of traces gives the risk assessor the opportunity to explore the database adequately for possible security properties. Also formal methods have played an important role in the analysis of security in information technology environments, notably are Z (Spivey, 1992) and BAN logic (Burrows et al., 1990) which are also based on logic. However these approaches do not focus on the *social* aspects of information security which requires emphasis on human factors and the need to model information flows and the interplay between various participants in the information system. The importance of an approach such as presented in this thesis which focuses on the human factors in the security of information systems is evident in studies by Whitman (2003) and CSI/FBI (Power, 2002) which ranks act of human error or failure (including accidents and employee mistakes) among the top threats identified for information systems. Security experts Peikari and Fogie (2003) state that *"For a policy to show any return on investment, it must become integrated into the processes and procedures of a business, along with the support of the people who are expected to follow it.*

Without this type of attention and support, a security policy will be worthless." Our model integrates security requirements into the processes and procedures and provide the mechanism for investigating security properties through querying. The results obtained may then be useful in determining the sufficiency of the security requirements thereby leading to a secure system design.

5.6 Summary of Chapter

This Chapter introduced the computational verification framework (CVF) for security requirements. The framework is inspired by the lack of computational means for verifying security requirements in the approaches for security requirements elicitation reviewed. CVF addresses this gap by providing a means of reasoning about security requirements through a computational model. In order to make it easier for modelling scenarios in InstAL, which abstracts the underlying reasoning logic Ans$Prolog$, CVF takes scenario modelled in UML activity diagrams and gives equivalent translation in InstAL. The details of how this translation is made possible is presented in the next chapter. However, using a scenario from a medical application, this chapter presents an overview of how one can reason about a system's security requirements. It starts with the modelling of the scenario in InstAL to the querying of the model, which is the means by which the reasoning can be done. The chapter concludes by discussing generalisation issues which include scalability/usability and validation of the framework. In the next chapter, we present the tools that further makes the framework usable. Particularly we address the issue of the semantics of translating UML activity diagrams to InstAL specification.

Chapter 6

Tool Set Development

6.1 Introduction

In the previous chapter we presented an overview of the process we are proposing for analysing security requirements using the institutional framework. We made use of a scenario from one of the iTrust Medical Record System's use cases as presented in Figure 5-3. The activity diagram is used to capture the work-flow for achieving the goal of making a patient referral. It is able to give us the kind of model required for us to work with, that is, one that has identifiable actions, actors, and a defined sequence of actors and systems interactions. This was then translated into the InstAL model based on the institutional framework. This is then translated into the computational model in ASP which produces the database of traces (answer sets) suitable for analysis. While these tools are existing tools which have been in use already, we realise that their use will require some technical knowledge (syntax and semantics) of these tools to be able to use them for tasks such as the translation of models into InstAL, production of query files in *AnsProlog* for analysis, and the understanding of the resulting answer sets. We have therefore built and extended some tools which will make the entire process more readily usable without necessitating learning much of the tools' technical details. We shall be presenting the tools and their extension in this chapter.

6.2 XML2InstAL: Activity diagram to InstAL Translator

The example InstAL models examined so far have been entirely written by hand. There is no tool through which system model specifications could be either totally or partially translated automatically into an InstAL specification. This implies that the usage of the tool has some cost overhead, that is a user has to learn the syntax and semantics of InstAL before the user can use it to model the desired system. In view of this barrier to adoption, the XML2InstAL translator was developed to ease the translation of mal-activity diagrams drawn in UML to InstAL specifications. Our translator makes use of an XML parser to parse the XML representation of UML activity diagram into an equivalent institutional model in InstAL.

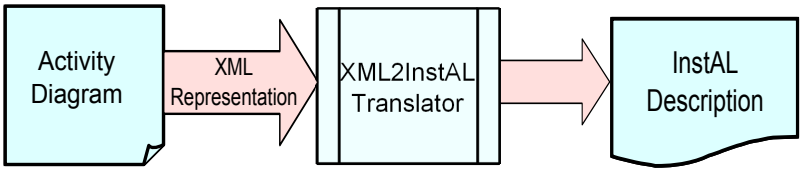


Figure 6-1: XML to InstAL Translation

XML2InstAL translator is developed in the Python programming language and uses the ElementTree XML API of the language to implement the parsing task.

6.2.1 Semantics of UML Activity Diagrams

We present here the notion of UML Activity Diagrams (ADs) (OMG, 2011) and describe the semantics needed for our translation of UML models to InstAL models.

Activity diagram is a directed graph of nodes and edges as in Table 6.1. The nodes are categorised into $ActivityNodes$, $ControlNodes$ and $ObjectNodes$. $ActivityNodes$ are further categorised into $actionNode$, $sendNode$ and $receiveNode$. There are two distinguished $ControlNodes$ namely $initialNode$ and $finalNode$ which are unique for each activity diagram. Each node is a member of some activity region of the AD called a $Partition$ (also known as swim lane). Partitions provide a

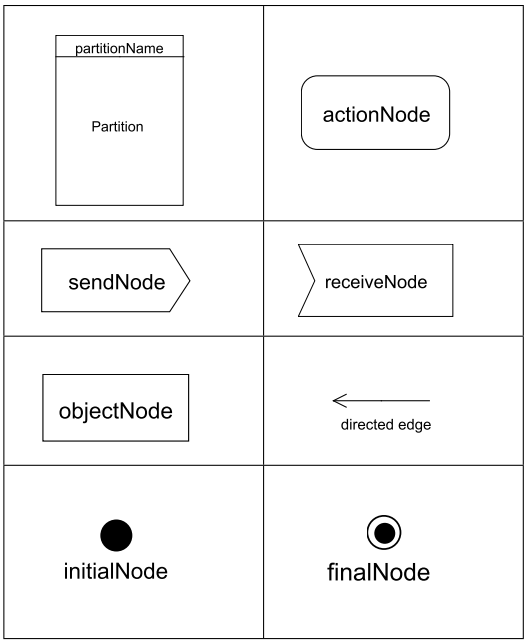


Table 6.1: UML symbols used

way to group activities performed by the same actor on an activity diagram. It also groups activities in a single thread. For instance activity partition P may be an entity of the system responsible for executing some set of actions. Partitions are identified by $partitionNames$. Nodes are linked to each other within and between activity partitions by directed edges. Each activity diagram starts with a single $initialNode$ and ends with a single $finalNode$, each of which may be in any of the activity partitions.

ADs are intended for modelling workflows. The suitability of using UML 2.0 ADs for modelling business processes has been investigated and presented in Russell et al. (2006); Wohed et al. (2006); Dumas and Hofstede (2001). Some of the points identified in support of UML 2.0 ADs with respect to alternative workflow modelling systems include the fact that they; *i)* offer comprehensive support for control-flow and data perspectives, *ii)* support signal sending and receiving at the conceptual level, and *iii)* provide a seamless mechanism for decomposing an activity specification into sub-activities. The combination of this decomposition capability with signal sending yields a powerful approach for handling activity interruptions. These features make UML activity diagrams suitable for use as a notation for graphical modelling of our workflow scenarios. Since our work is

not focused on UML modelling processes or workflows, we do not intend to dwell on the issues of which notation is better for modelling workflows and business processes among the currently researched and used notations which include UML Activity Diagrams (OMG, 2011; Russell et al., 2006), the Business Process Modelling Notation (BPMN) (OMG, 2011; Wohed et al., 2006), Event-driven Process Chains (EPCs)(van der Aalst, 1999; Scheer et al., 2005), Workflow nets (van der Aalst, 1998; Karniel and Reich, 2011), and the Business Process Execution Language (BPEL) (Farahbod et al., 2004).

As suitable as UML activity diagrams may be for modelling computational and business/organisational processes (Wohed et al., 2006), it lacks a formal semantics for analysis. This lack of formal semantics has inspired a number of research works (such as Eshuis and Wieringa (2001); Lam (2007); Knieke and Goltz (2010); Bouabana-Tebibel and Belmesk (2007); Guelfi and Mammar (2005); Yizhi et al. (2004); Störrle (2004); Vitolins and Kalnins (2005); Xu et al. (2008)) aimed at formalising the semantics of UML activity diagrams for various purposes based on the informal semantics provided by the Object Management Group (OMG)(OMG, 2011). In the current version of UML 2.0 the informal semantics of activities is based on Petri nets, therefore the attempts at formalizing the semantics focus on token movement. We are more interested however in actors, the events associated with actors, and the transition of events in a use case, which the reviewed literature on AD semantics formalisation does not directly address. Hence we do not find these formalisations entirely useful for our work. However, we found the works in Bouabana-Tebibel and Belmesk (2007) and Xu et al. (2008) more helpful in the sense that they both present formal definitions for the basic notations of activity diagrams and make attempts at grouping activities into partitions even though this was not formalised. We therefore adapt the definitions in these references and present a formal semantics of activity diagrams starting with definition of our notations.

We adopt UML 2.0 as a pragmatic choice in that the notation is widely understood, even if it lacks an agreed semantics and because at least the intention expressed in a specification is broadly accessible.

In order to formally define our activity diagram, we define the components (see

Table 6.1) that make up our activity diagram as follows;

Definition 17 (Components of an activity diagram)**.** *We denote the components as follows;*

I *- initial node*
F *- final node*
$A = \{a_1, a_2, \ldots, a_m\}$ *- finite set of action nodes*
$O = \{o_1, o_2, \ldots, o_m\}$ *- finite set of object nodes*
$S = \{s_1, s_2, \ldots, s_m\}$ *- finite set of send nodes*
$R = \{r_1, r_2, \ldots, r_m\}$ *- finite set of receive nodes*
$E = \{e_1, e_2, \ldots, e_m\}$ *- finite set of edges*
where $A \cap O \cap S \cap R \cap E = \emptyset$ *and the value of* m *is different in each case.*

From these components we define the following compositions;

- $V = A \cup S \cup R$ *is the set of activity nodes*
- $C = \{I, F\}$ *is a set of control nodes*
- $N = V \cup C \cup O$ *is the set of all nodes*

Since these components are members of some activity regions of the activity diagram called a partition, we define the activity diagram in terms of partitions as follows;

Definition 18 (Activity Diagram)**.** *An activity diagram is a set of activity partitions* $P = \{p_i \ldots p_m\}$*. Each partition is a tuple* $p_i = \langle X_i, E_i, pname \rangle$ *where* $X_i \subseteq N$ *and* $X_i \cap X_j = \emptyset$*, for* $i, j \in [1, \ldots, m]$*,* $i \neq j$*. pname is an attribute of the partition which specifies the name of the partition.*

In activity diagrams, edges provide the means of transition between nodes within a partition and also between partitions. We would want to be able to identify which edges go into which nodes and which edges come out of which nodes. In this way we can relate actors to actions. Definition 19 presents the categorisation of edges;

Definition 19 (Categorisation of edges)**.** *We define the out-going edge of a node by the function* $e_{out} : N \rightarrow E$ *such that for any node* $n \in N$ *and edge* $e \in E$*, we have* $e_{out}(n) = e$ *where* $n \neq F$*. Similarly, we define the in-coming edge of a node by the function* $e_{in} : N \rightarrow E$ *such that* $e_{in}(n) = e$ *where* $n \neq I$*.* $e_{in}(I) = e_{out}(F) = \emptyset$*.*

Each activity diagram consists of a single initial node I and a single final node F which indicate the starting and the ending points of the scenario being modelled. Therefore the initial node does not have an in-coming edge and the final node does not have an out-going edge.

Definition 19 relates edges and nodes. However we also want to be able to relate partitions, since partitions are groups of events that an actor enacts. We need to be able to identify the transition from one partition to another. We therefore define functions on edges and partitions that describes this relationship as follows;

Definition 20 (Source and Target partitions)**.** *For any edge* $e \in E$*, let* $p_t, p_s \in P$ *denote target partition and source partition respectively. For any partition* $p \in P$ *and node* $n \in N$ *we define the functions* $TarP : E \times N \rightarrow P$ *and* $SrcP : E \times N \rightarrow P$ *such that;*

- $TarP(e, n_1) = p_t$ *s.t.* $e_{out}(n_1) = e_{in}(n_2) = e, n_1 \in p, n_2 \in p_t$ *defines the target partition of an out-going edge* e *from a node* n_1 *and*
- $SrcP(e, n_2) = p_s$ *s.t.* $e_{in}(n_2) = e_{out}(n_1) = e, n_1 \in p_s, n_2 \in p$ *defines the source partition of an in-coming edge* e *into a node* n_2*.*

In the next section we make use of these definitions to formalise the translation of our UML models to InstAL.

6.2.2 Soundness of activity diagrams

According to the UML specification (OMG, 2011), activities have Petri net-like semantics which is based on token flow. This implies that when an activity is executed, the initial node creates a token which is then routed through the activity

workflow and finally consumed by the final node of the workflow. The token here corresponds to the case that is being handled by the workflow which starts at the initial node and terminates at the final node. We discuss the soundness of activity diagrams based on the soundness properties of workflows as described by van der Aalst (van der Aalst, 1997; van der Aalst and van Hee, 2004) and van Hee (van Hee et al., 2003). They state that every workflow should fulfil the following basic requirements:

R1: The workflow should have well-defined pre and postconditions.
R2: The workflow should not contain any useless elements.
R3: If the end condition is reached, no more tasks should be processed.
R4: The end condition should finally be reached.

We establish a correspondence betwen these generic workflow requirements and activity diagrams as follows:

AD1: The activity must have exactly one initial node and final node
AD2: Every activity node must be associated with a partition
AD3: The last node of a transition sequence must be the final node
AD4: A transition sequence from any activity node should lead to the final node.

Taking into account these requirements, we define the soundness property of Activity Diagrams as follows;

Given an activity diagram with a set of nodes $n_1, \ldots, n_m \in N$, an initial node I, final node F, and edges $e_1, \ldots, e_k \in E$ we define the following notations;

- $n_1 \xrightarrow{e_i} n_2$ is a *transition* from a node n_1 to node n_2 by the edge e_i
- A node n_m is *reachable* from I denoted as $I \xrightarrow{*} n_m$ if and only if there is a transition by the sequence of edges $\langle e_1, \ldots, e_{m-1} \rangle, e_i \in E$ such that $I \xrightarrow{e_1} n_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} n_m$.

Based on van der Aalst and van Hee (2004, p. 270) and following the correspondence AD1 - AD4 we formally define the soundness property of an activity diagram as follows;

Definition 21 (**Soundness**). *A scenario modelled by an activity diagram AD as defined in Definition 18 and having one initial node I and one final node F, is sound if and only if:*

> *For every activity node $v \in V$ reachable from node I, there exist transition steps leading from node v to final node F. Formally:*
>
> $\forall v \in V : I \xrightarrow{*} v \xrightarrow{*} F$

6.2.3 Translation of UML Activity Diagram to InstAL

We use the UML diagram tool called Visual Paradigm for UML[1]. This tool represents details of the features used in a diagram in XML format. The ability for us to identify activity partitions (swim lanes) is particularly important for us. We use the UML symbols in Table 6.1 from the Activity Diagram function of the program.

In this section we outline the process for translating UML activity diagrams to InstAL specifications. An InstAL specification consist of declarations for *type*, *events*, $fluents$, and *rules* as specified in Section 4.6.1. In order to be able to derive expressions for these InstAL features, we make the following associations between the UML diagram symbols and InstAL:

i) **Partition:** each partition translates to a `type` in InstAL. Types, which are declared as `type` $identifier$ in InstAL, establish finite value sets of disjoint monomorphic types[2] whose instances are specified in a domain file. An example of a domain file is given in Figure 5-10 on page 121. The types in the domain file are specified as $type-identifier : type-instance(s)$ with the instances being separated by spaces. For example `HCP: sHCP rHCP` (Figure 5-10, line 1). The Answer Set Solver will then substitute those values (e.g. `sHCP` and `rHCP`) whenever the type `HCP` is specified for the events and fluents.

   We associate the partition name *partitionname* with the $identifier$. There-

[1] Visual Paradigm for UML is a commercial tool but there is a free community edition for non-commercial use which is what we used. Available here: http://www.visual-paradigm.com/download/vpuml.jsp?edition=ce

[2] Every expression has exactly one type

fore our translation is of the form `type` *partitionname*.

ii) **Activity nodes (action, send, receive nodes):** each of these translates to an InstAL `event` declaration of the form `exogenous event` *eventname*(*type*$^+$) for physical world events, for example `exogenous event loginChal(System, HCP)` in Figure 5-5, line 9. Similarly, institutional events are declared as `inst event i`*eventname*(*type*$^+$) for example `inst event iloginChal(System,HCP)` in Figure 5-5, line 26. In our translation we associate activity nodes with *eventname* and partition name with *type* resulting in the expression *activitynode*-(*partitionname*$^+$) which we use for translating exogenous and institutional events.

iii) **Object node:** the occurrence of events in a system brings about changes in the state of the system. The facts that may be brought about in the system state are known as *inertial fluents* and are declared in InstAL in the form `fluent` $fluentname(type)$, for example `fluent policyApplied(HCP)` in Figure 5-6, line 52. In our translation, object node corresponds to $fluentname$ hence we express inertial fluents from object nodes of the activity diagram as *objectnode*(*partitionname*).

iv) **Directed edges:** these do not have an explicit representation in InstAL but they are only used in the translation process to express relations between partitions in situations where the enactment of an event involves correspondence between two partitions.

v) **Control nodes (initial and final nodes):** Like directed edges, these only provide information for determining the beginning and the end of a transition. They do not have an explicit representations in InstAL.

6.2.3.1 *Translation Semantics*

Following our definitions in Section 6.2.1, we proceed by setting out the basis for translation of our UML based models to InstAL specifications which we illustrate in Section 6.2.3.2. We present the various translation notations that we use for our translations as follows;

Given an activity diagram with the name *ADname*, we define the following trans-

lation notation:

$Tr(ADname) \rightarrow$ `institution` $ADname$

This translates the activity diagram name into the name of the InstAL institution model.

For each partition $p \in P$ the following translations hold:

i) The partition name translates to the **type declaration** of the InstAL institution as follows;

$Tr(pname) \rightarrow$ `type` $pname$;

ii) An activity node $v \in V$ translates as;

$Tr(v) \rightarrow v(pname)$

iii) For every send node $s \in S$ and receive node $r \in R$ the following translations hold:
$Tr(s) \rightarrow s(p_i name, p_j name), i < j$
$Tr(r) \rightarrow r(p_j name, p_i name), i < j$

This is to say that every send node and every receive node is represented as an InstAL specification of the form $eventname(type, type)$.

Items ii and iii represent translations for declaring events as described in Section 6.2.3 item ii. For instance for each activity node $v \in V$ in each partition, exogenous events are declared as follows;

`exogenous event` $Tr(v)$`;`

Similarly, institution events are declared as follows;

`inst event` $iTr(v)$`;` In order to differentiate institution events from exogenous events, we prefix the translation of the activity nodes with i.

iv) Every object node $o \in O$ translates to the following;

$Tr(o) \rightarrow o(pname)$

This translation is used in the declaration of fluents as described in Section

6.2.3 item iii. Therefore for each object node $o \in O$ we generate fluent declarations as:

`fluent` $Tr(o)$`;`

Rule Definitions: In order to be able to explore the state space of a model as obtained in Petri Nets (Lakos and Petrucci, 2007; Kristensen, 2010; Camilli, 2012), InstAL specification provides a number of rules that realise the execution mechanism needed to make state exploration possible. The rules which are described in detail in Section 4.6.1 are *consequence rules*, *generation rules*, and *initiate rules*. These rules establish the presence of physical world events and the dynamics of how institutional events and fluents are brought about. We present here how the declarations for these rules can be achieved from our translations as follows;

(a) **Generation rules:** Events in the real world generate corresponding events in the institutional framework which in turn result in the changing of institutional fluents. A Generation rule, represented as $triggerEvent$ `generates` $institutionalEvent$ [$condition$], describes when institutional events may be generated subject to an optional condition. We derive events from activity nodes v, hence generation rules are translated as;

$Tr(v)$ `generates` $iTr(v)$`;`

(b) **Consequence rules:** Consequence rules, which describe when fluents change in response to the occurrence of events, are declared as $triggerEvent$ `initiates` $institutionalFluent$ [$condition$]. The condition which is optional describes the fluents which may be true in order for the rule to have an effect. For example the rule `icheckRef(Checker) initiates goodRef(HCP)` in Figure 5-8 line 106 adds the fluent `goodRef(HCP)` to the institutional framework. Since we derive fluents from object nodes, if the target node of an out-going edge from an activity node $v \in V$ is an object node $o \in O$ a consequence rule is generated as;

$iTr(v)$ `initiates` $Tr(o)$`;`

(c) **Initiate rules:** These describe the state of the institution whenever the in-

stitution is created. Using the translations for permission and empowerment, we generate the initial rules as follows;

`initially perm(`$Tr(v)$`), perm(`$iTr(v)$`), pow(`$iTr(v)$`);`

Soundness and Completeness of translation: We present the soundness and completeness of our translation mechanisms as follows;

Definition 22. *Soundness of translation: For every activity diagram AD, the translation is sound if and only if:*

(i) For every partition $p \in P$ *there is a translation;* $\forall\ p \in P\ \exists\ q\ s.t.\ Tr(p) = q$

(ii) There must be a translation for every activity and object nodes i.e. $\forall\ x \in V \cup O\ \exists\ t\ s.t.\ Tr(x) = t$

In our framework the completeness property ensures that each UML activity symbol defined can be mapped to an InstAL specification. Our translation process is complete since in Section 6.2.3.1 we have defined a rule for each element of an activity diagram used in our model. Table 6.2 presents a summary of the translation of UML activity diagrams (AD) into InstAL elements and how they may be used in the InstAL specification. In the next section we present illustration of the models translated from UML activity diagrams based on the semantics presented here.

UML-AD Element	**InstAL Translation**	
	InstAL Element	**Example Usage in InstAL Structure**
AD name (ADname)	$ADname$	`institution` $ADname$;
Partition name (pname)	$pname$	`type` $pname$;
Activity node (v)	$v(pname)$	`exogenous event` $v(pname)$; `inst event` $iv(pname)$;
Send node (s)	$s(p_iname, p_jname), i < j$	`exogenous event` $s(p_iname, p_jname)$; `inst event` $is(p_iname, p_jname)$;
Receive node (r)	$r(p_jname, p_iname), i < j$	`exogenous event` $r(p_jname, p_iname)$; `inst event` $ir(p_jname, p_iname)$;
Object node (o)	$o(pname)$	`fluent` $o(pname)$;

Table 6.2: Summary of UML-InstAL Translation

6.2.3.2 Example using an Examination Paper Scenario

We illustrate how the activity diagram translates into an InstAL specification with an example scenario. Figure 6-2 shows an scenario involving three actors in designated activity partitions.

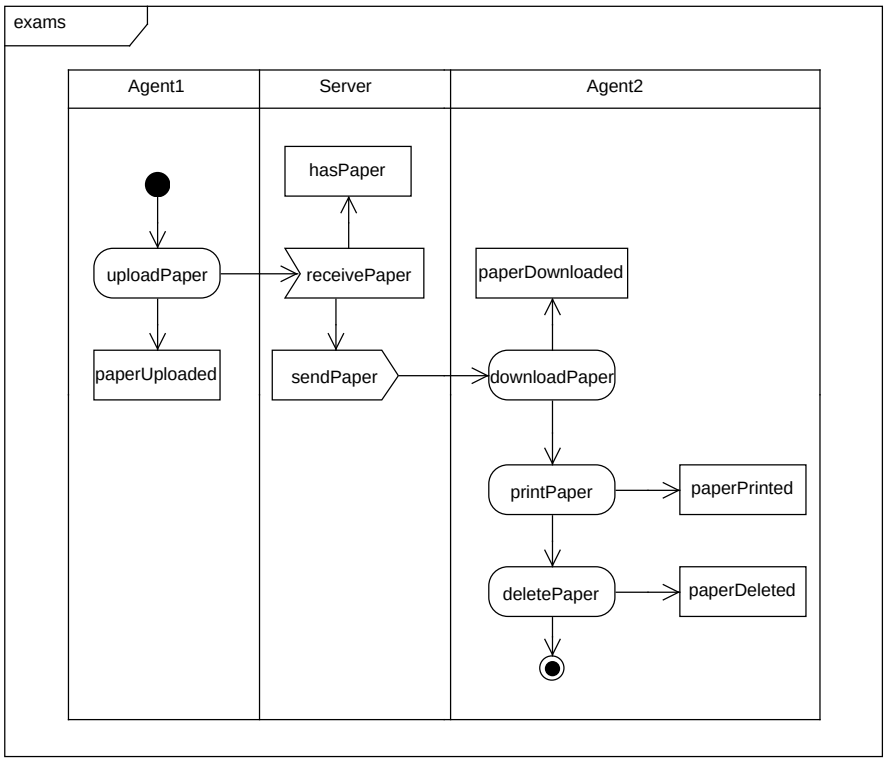


Figure 6-2: A simple example of an Activity Diagram

This activity diagram captures a simple scenario where two actors `Agent1` and `Agent2` are interacting with another component of the system `Server` to produce printed copies of an examination paper. To keep it simple for the purpose of illustration, we are not presenting the complete process here. In this excerpt, `Agent1` uploads the paper to the `Server`. `Agent2` downloads the paper from the `Server`, prints the paper, and deletes the file from its own computer.

Notice here that we did not represent the attacker or the attacker activities here. This is in view of our earlier stated assumption in Section 5.3.2 that an attack could happen from any point of vulnerability in the scenario. We are

therefore not explicitly modelling the attack scenarios but the user actions and interactions.

Following the InstAL specification generation semantics presented in Section 6.2.3.1, we illustrate how the InstAL specifications can be generated.

Type declarations: The type declarations for the institutional model is generation from each of the activity partitions labelled $Agent1$, $Server$, and $Agent2$ as follows;

Using $Tr(p) \rightarrow \texttt{type}\, p$; we have the following generated;

```
type Agent1;
type Server;
type Agent2;
```

Events declarations: Events declarations are generated from each of the activity nodes.

From translation (ii) in Section 6.2.3.1 we have the following translations of each activity node v in each activity partition p based on the translation $Tr(v) \rightarrow v(p)$.

From partition $Agent1$;

```
uploadPaper(Agent1);
```

From partition $Server$;

The *send* and *receive* nodes in this partition are translated based on item (iii) in Section 6.2.3.1 giving the following;

```
receivePaper(Server, Agent1);
sendPaper(Server, Agent2);
```

From partition $Agent2$;

```
downloadPaper(Agent2);
printPaper(Agent2);
deletePaper(Agent2);
```

Declarations for exogenous and institutional events are generated as follows;

```
exogenous event uploadPaper(Agent1);
exogenous event receivePaper(Server, Agent1);
exogenous event sendPaper(Server, Agent2);
exogenous event downloadPaper(Agent2);
exogenous event printPaper(Agent2);
exogenous event deletePaper(Agent2);

inst event iuploadPaper(Agent1);
inst event ireceivePaper(Server, Agent1);
inst event isendPaper(Server, Agent2);
inst event idownloadPaper(Agent2);
inst event iprintPaper(Agent2);
inst event ideletePaper(Agent2);
```

Fluent declarations: Fluents are translated from the object nodes in the UML model. Based on item (*iv*) in Section 6.2.3.1 the object nodes in each of the partitions are translated to give the following;

```
paperUploaded(Agent1);
hasPaper(Server);
paperDownloaded(Agent2);
paperPrinted(Agent2);
paperDeleted(Agent2);
```

The fluent declarations are generated as follows;

```
fluent paperUploaded(Agent1);
fluent hasPaper(Server);
fluent paperDownloaded(Agent2);
fluent paperPrinted(Agent2);
fluent paperDeleted(Agent2);
```

Permission and empowerment: Based on the translation of events from activity nodes, permissions are declared for all events with additional *power* declarations for institutional events. We therefore have the following declarations;

```
perm(uploadPaper(Agent1));
perm((Server, Agent1));
perm(sendPaper(Server, Agent2));
perm(downloadPaper(Agent2));
perm(printPaper(Agent2));
perm(deletePaper(Agent2));
```

```
perm(iuploadPaper(Agent1));
perm(ireceivePaper(Server, Agent1));
perm(isendPaper(Server, Agent2));
perm(idownloadPaper(Agent2));
perm(iprintPaper(Agent2));
perm(ideletePaper(Agent2));
pow(iuploadPaper(Agent1));
pow(ireceivePaper(Server, Agent1));
pow(isendPaper(Server, Agent2));
pow(idownloadPaper(Agent2));
pow(iprintPaper(Agent2));
pow(ideletePaper(Agent2));
```

Rules declarations: InstAL specifications also consist of three rules as presented in Section 4.6.1. These are *consequence rules* which describe fluent changes as events occur, *generation rules* which describe the generation of institutional events and *initial rules* which describe the state of the institution at the creation of the institution. These rules are generated as follows;

(a) Generation rules: The following rules are generated;

```
uploadPaper(Agent1) generates iuploadPaper(Agent1);
receivePaper(Server, Agent1) generates ireceivePaper(Server, Agent1);
sendPaper(Server, Agent2) generates isendPaper(Server, Agent2);
downloadPaper(Agent2) generates idownloadPaper(Agent2);
printPaper(Agent2) generates iprintPaper(Agent2);
deletePaper(Agent2) generates ideletePaper(Agent2);
```

(b) Consequence rules: The following rules are generated;

```
iuploadPaper(Agent1) initiates paperUploaded(Agent1);
ireceivePaper(Server,Agent1) initiates hasPaper(Server);
idownloadPaper(Agent2) initiates paperDownloaded(Agent2);
iPrintPaper(Agent2) initiates paperPrinted(Agent2);
ideletePaper(Agent2) initiates paperDeleted(Agent2);
```

The equivalent InstAL specification for the model in Figure 6-2 consists of the declarations presented in Figure 6-3.

```
institution exams;

type Agent1;
type Server;
type Agent2;

exogenous event uploadPaper(Agent1);
exogenous event receivePaper(Server, Agent1);
exogenous event sendPaper(Server, Agent2);
exogenous event downloadPaper(Agent2);
exogenous event printPaper(Agent2);
exogenous event deletePaper(Agent2);

inst event iuploadPaper(Agent1);
inst event ireceivePaper(Server, Agent1);
inst event isendPaper(Server, Agent2);
inst event idownloadPaper(Agent2);
inst event iprintPaper(Agent2);
inst event ideletePaper(Agent2);

fluent fpaperUploaded(Agent1);
fluent fhasPaper(Server);
fluent fpaperDownloaded(Agent2);
fluent fpaperPrinted(Agent2);
fluent fpaperDeleted(Agent2);

uploadPaper(Agent1) initiates fpaperUploaded(Agent1);
receivePaper(Server,Agent1) initiates fhasPaper(Server);
downloadPaper(Agent2) initiates fpaperDownloaded(Agent2);
PrintPaper(Agent2) initiates fpaperPrinted(Agent2);
deletePaper(Agent2) initiates fpaperDeleted(Agent2);

uploadPaper(Agent1) generates iuploadPaper(Agent1);
receivePaper(Server, Agent1) generates ireceivePaper(Server, Agent1);
sendPaper(Server, Agent2) generates isendPaper(Server, Agent2);
downloadPaper(Agent2) generates idownloadPaper(Agent2);
printPaper(Agent2) generates iprintPaper(Agent2);
deletePaper(Agent2) generates ideletePaper(Agent2);

Initially perm(uploadPaper(Agent1)),
perm((Server, Agent1)),
perm(sendPaper(Server, Agent2)),
perm(downloadPaper(Agent2)),
perm(printPaper(Agent2)),
perm(deletePaper(Agent2)),
perm(iuploadPaper(Agent1)),
perm(ireceivePaper(Server, Agent1)),
perm(isendPaper(Server, Agent2)),
perm(idownloadPaper(Agent2)),
perm(iprintPaper(Agent2)),
perm(ideletePaper(Agent2)),
pow(iuploadPaper(Agent1)),
pow(ireceivePaper(Server, Agent1)),
pow(isendPaper(Server, Agent2)),
pow(idownloadPaper(Agent2)),
pow(iprintPaper(Agent2)),
pow(ideletePaper(Agent2));
```

Figure 6-3: The translated InstAL model using XML2InstAL translator.

The translator is able to give us a description of the model of the system with the intended users interacting with the aim of achieving a particular goal. We are not assuming any particular attacker in our approach, although the focus is on insider attacks which are due to the interaction of actors and the environment. As a result, the model generated initially grants permission to all exogenous events. Also associated institutional events are also permitted and empowered initially. This is to grant a form of autonomy of actions to the actors. With this, it would allow us to investigate on the kind of security vulnerabilities that could arise as the actors interact in the life-cycle of the scenario.

With the addition of the automatic InstAL specification generator, we now modify the InstAL translation process (Figure 4-8 on page 98) originally proposed in Cliffe (2007) as shown in Figure 6-4.

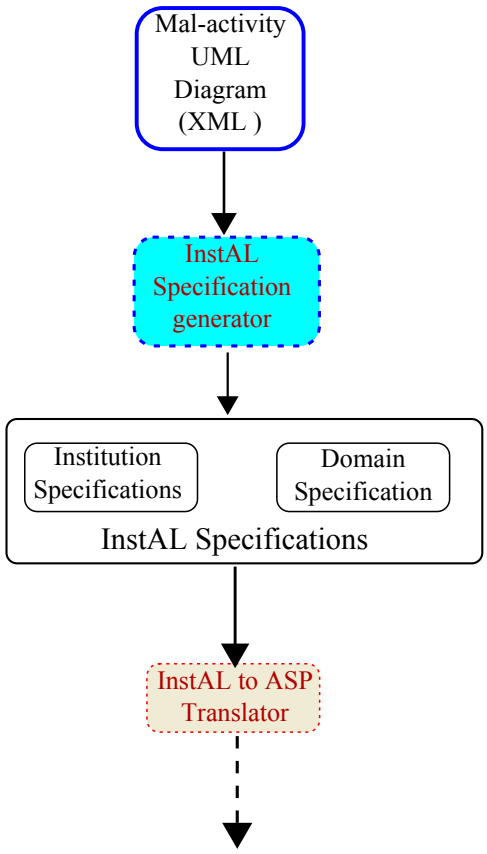


Figure 6-4: Modification of InstAL Translation Process by addition of the InstAL specification generator.

We would also like to note that at this point of writing this thesis, there is no unified schema for UML to XML translation. As a result, no two UML diagram tools would generate the same XML files. This implies that our translator would only work without any modification if the XML source file is generated by the same UML diagram tool we used (Visual Paradigm for UML). In the future, we aim at standardizing this tool so that it can accept any standard XML file.

6.3 *pyinstql*: Semi-automatic Query Translator

Institutional frameworks make it possible to monitor the permissions, empowerment and obligations of participants and to indicate violations when regulations (security requirements in our case) are not followed or when security properties are not satisfied. The change of the state over time as a result of actors' actions provides traces from which an institution designer can query and verify properties, their effects and expected outcomes in an institution. Institutions are useful when particular properties can be verified as satisfying all possible scenarios.

One of the major goals of this thesis to be able to perform verification and analysis of security properties by querying the "database" of traces produced from the institutional model. InstAL, an action language for modelling institutions, provides an institution designer with a useful abstraction away from the underlying *AnsProlog* specification. This level of abstraction is however lost when it comes to specifying queries about an institution. Since querying is a vital part of the modelling process through which desired models (answer sets) may be generated, queries have to be written directly in *AnsProlog*. It means that the designer or user is required to know the semantics of InstAL to *AnsProlog* translation in order to write queries for the InstAL specification. This is undesirable for effective use of the tool by domain experts. Particularly in our case, while we model system work-flows and processes in InstAL through the activity/mal-activity diagrams, we analyse the security state of the system by querying the institution based on the security properties that we are investigating. There is therefore a need for a way of expressing the queries in a more natural language for effective querying of the institution. By this we further modify the current InstAL development process (Figure 4-8 on page 98) by the addition of a layer which produces

the query program. This is illustrated in Figure 6-5 where the additional features are indicated by blue colour.

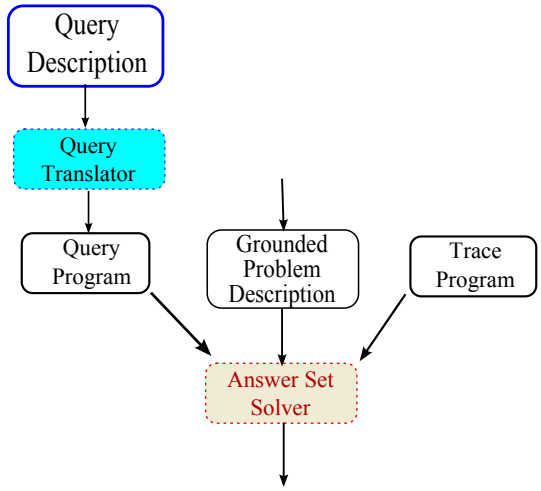


Figure 6-5: Modification of InstAL Translation Process by addition of the Query Translator.

6.3.1 Query Components

We aim at providing a query language that is usable to a user who has little or no knowledge of the *AnsProlog* syntax. Our query file consist of the following components;

- **Facts:** These express the facts that we want to sustain in the traces. Importantly we would like to limit our traces to those which concern specific exogenous events occurring at specific instants. Our query file will therefore express the fact `observed(event,instant)`.
- **Rules:** Rules express the conditions and constraints which describes the properties we are investigating.
- **Flags:** Answer sets usually consists of many information for every trace produced. These would include fluents, events, and other trace information. The flags `#hide` is used to hide all the traces in an answer set while the flag `#show` $artefact$ serves like a filter which displays only the traces specified as the $artefact$. For example `#show holdsat(F,I)` will only display fluents `F` which hold at each instant `I` of the trace.

There has been a previous attempt at developing a query language for InstAL called InstQL by Hopton et al. (2010). However we did not find InstQL too useful for a number of reasons. Some of these include;

i.) It handles only simple predicates with single argument. However we are dealing with predicates involving multiple arguments.

ii.) It cannot produce complete query specification that meets our requirements as described earlier. By this we mean facts such as "`observed(`*event*`,`*instant*`).`" This rule is necessary particularly for the purpose of validation. It limits traces only to those which concern the stated observed exogenous events. Also there is no provision for the flags "`#hide`" and "`#show` *artefact*".

iii.) It cannot handle queries expressed as multiple disjunctions or conjunctions of events or fluents.

iv.) InstQL had problems with handling time instants for the query fact `after(`*instant*1`,` *instant*2`)` which interprets as the time instant *instant*1 precedes *instant*2 for any associated event or fluent.

Based on these reasons, we could not use InstQL and therefore had to develop our query translator - *pyinstql* - in a language (Python) that would make it easy to maintain and improved upon with time.

6.3.2 *pyinstql* Syntax

In this work we aim at analysing security properties from users' perspective. We are therefore concerned with the occurrence of events and the effects of users' interactions (modelled as events) in terms of the state of the system with respect with the fluents that hold at specific instants. It is also important for us to know what violations occur and the instant at which they occur. This information will give the domain experts insight into how effective the security requirements would be.

Based on these, the *pyinstql* queries are formed based on the following predicates:

`happens(Event,Instant)` is used for querying the occurrence of an event. If a specific instant is specified, we would be querying the occurrence of the specified

event at the specified instant else all instants would be considered. For example: `happens(sendPaper(lecturer,hod),2)` would query for the occurrence of the event `sendPaper(lecturer,hod)` at time instant `2`. This predicate can be used on its own or can be used as a condition for determining other properties.

`holds(Fluent,Instant)` is used in a similar way with `happens` but it handles fluents. It will query for the existence of fluents at specified instants.

`violates(Event,Instant)` handles the occurrence of violation events.

It is possible to specify these predicates without specifying the instants. In this case it means the event should occur and the fluent would be true respectively at some point in the life time of the institution. This is expressed as follows:

```
<predicate> ::= happens( <identifier> ) | holds(<identifier>) |
                violates(<identifier>)
<identifier> ::= <name> | <name><param_list>
```

where *identifier* corresponds to an event, a fluent, or an instant.

The unary operator `not` provides negation (as failure) expressed as follows:

```
<literal> ::= not <predicate> | <predicate>
```

Sub-queries (sub-conditions) are also supported by *pyinstql*. For instance we can define a condition say `this_cond` specifying some desired property. This sub-condition can then be joined to some other criteria, for example "`this_cond and holds(f)`". We reference sub-conditions within rules as *condition literals* as follows:

```
<condition_literal> ::= not <identifier> | <identifier> |
                        <identifier> ( <identifier> , <identifier> )
```

Query conditions are built of *terms* expressed as:

```
<term> ::= <after_expr> | <condition_literal>
```

The after expression allows for construction of `<while_expr>` as:

```
<after_expr> ::= <while_expr> | <while_expr> after <after_expr>
```

and the `while` expression further allows for simpler constructs of `<literal>`:

```
<while_expr> ::= <literal> | <literal> while <while_expr>
```

The connectives **and** and **or** provide logical conjunction and disjunction which may be used to group terms:

```
<conjunction> ::= <term> and <conjunction> | <term>
<disjunction> ::= <term> or <disjunction> | <term>
```

In order for us to be able to create arbitrary combinations of predicates and named conditions together with the logical operators `and, or, not`, we need to make conditions declaration constructed in this form:

```
<condition_decl> ::= condition <identifier> : <disjunction>
                     | condition <identifier> : <conjunction>;
```

This allows us to use the `condition name` as a `condition_literal`.

Constraints specify the properties of the trace that must be true. We can also specify constraint properties in *pyinstql* as follows:

```
<constraint> ::= constraint <disjunction> | <conjunction> ;
```

In order to be able to specify the particular events for which verification is being carried out and to constrain the answer sets to show specific results, we also provide expressions for `observe` and `show` as follows:

```
<observe> ::= observe <param_list>;
<show> ::= show <param_list>;
```

Table 6.3 on the next page shows a summary of the *pyinstql* syntax.

6.3.3 *pyinstql* Semantics

A *pyinstql* query is composed of semi high-level language statements and the semantics is defined by the translation function $Trans$ which translates the *pyinstql* query statements into *AnsProlog* query specifications consisting of *AnsProlog* rules. This is typically a singleton set i.e each statement generates only one rule except expressions involving disjunctions.

Expression		Definition
< variable >	::=	[A − Z][a − zA − Z0 − 9]∗
< variable_list >	::=	< variable >, < variable_list > | < variable >
< name >	::=	[A − Z][a − zA − Z0 − 9]∗
< param_list >	::=	(< variable_list >)
< identifier >	::=	< name >< param_list > | < name >
< predicate >	::=	happens(< identifier >)|holds(< identifier >)
< literal >	::=	not < predicate > | < predicate >
< while_literal >	::=	< literal > | < condition_literal >
< while_expr >	::=	< while_literal > while < while_expr > | < while_literal >
< after >	::=	after(< integer >)|after
< after_expr >	::=	< while_expr >< after >< after_expr > | < while_expr >
< condition_literal >	::=	not < identifier > | < identifier >
< term >	::=	< after_expr > | < condition_literal >
< conjunction >	::=	< term > and < conjunction > | < term >
< disjunction >	::=	< term > or < disjunction > | < term >
< conditiondecl >	::=	condition < identifier >:< disjunction >; | condition < identifier >:< conjunction >;
< constraint >	::=	constraint < disjunction >; | constraint < conjunction >;
< observe >	::=	observe < param_list >;
< show >	::=	show < param_list >;

Table 6.3: Summary of *pyinstql* Syntax

The semantics of predicates are defined by the following translations:

$$Trans(\texttt{happens(e)}) = \texttt{occurred(e, I), event(e), instant(I)} \quad (22.1)$$

$$Trans(\texttt{holds(fluent)}) = \texttt{holdsat(f, I), ifluent(f), instant(I)} \quad (22.2)$$

$$Trans(\texttt{violates(e)}) = \texttt{occurred(viol(e, I)), event(e), instant(I)} \quad (22.3)$$

Literals of the form `not P` where `P` is a predicate have the following semantics:

$$Trans(\texttt{not } P) = \texttt{not } Trans(\texttt{P})$$

Conjunction of terms takes the form:

$$Trans(c_1 \texttt{ and } c_2 \texttt{ and} \ldots \texttt{and } c_n) = Trans(c_1), Trans(c_2), \ldots, Trans(c_n)$$

while a disjunction which translates to more than one rule takes the following semantics depending on whether it is part of a condition declaration or a constraint:

$$\begin{aligned} Trans(\texttt{condition conditionName} \;:\; & c_1 \texttt{ or } c_2 \texttt{ or} \ldots \texttt{or } c_n;) = \\ & \{\texttt{conditionName} \leftarrow Trans(c_i).|1 \leq i \leq n\} \\ Trans(\texttt{constraint} \quad\;\; & c_1 \texttt{ or } c_2 \texttt{ or} \ldots \texttt{or } c_n;) = \\ & \{\texttt{newName} \leftarrow Trans(c_i).|1 \leq i \leq n\} \cup \\ & \{\bot \leftarrow \texttt{not newName}.\} \end{aligned}$$

The term `newName` denotes any unique *AnsProlog* identifier within the *AnsProlog* program that is the combination of the query and the action program. This atom becomes true if one of the sub-queries in the disjunction becomes true. Also each time instant `I` generated in the translation of a predicate a name for a unique time instant in the *pyinstql* query.

The semantics for `while` expression is:

$$Trans(L_1 \texttt{ while } L_2 \texttt{ while} \ldots \texttt{while } L_n) = Trans(L_1), Trans(L_2), \ldots, Trans(L_n), \texttt{instant(I)}$$

and the binary operator `after(n)` takes the form

$$Trans(W_i \texttt{ after(n) } W_3) = Trans(W_i), Trans(W_j), \texttt{after}(\texttt{t}_\texttt{i}, \texttt{t}_\texttt{j}, \texttt{n})$$

with $\texttt{t}_\texttt{i}$ and $\texttt{t}_\texttt{j}$ being the time instants generated by W_i and W_j respectively and n > 0.

Semantics for `observed` expression is defined as:

$$Trans(\texttt{observe e}) = \texttt{observed}(\texttt{e}, \texttt{I}).$$

This translates an $e \in \mathcal{E}_{ex}$ to a fact expressed as a predicate with the event `e` and time instant `I` as arguments.

In the case of a conjunction, events are separated by a comma and each event is translated in a new line. Time instants are also automatically allocated in increasing order beginning with `0` for the first event.

$$Trans(\texttt{observe e}_1, \ldots, \texttt{e}_n) = \begin{cases} \texttt{observed(e}_1, \texttt{0).} \\ \vdots \\ \texttt{observed(e}_n, \texttt{n} - \texttt{1).} \end{cases}$$

The `show` expression takes parameters which are `events`, `states`, and `violations`.

$$\begin{aligned} Trans(\texttt{events}) &= \texttt{occurred(e, I)} \\ Trans(\texttt{states}) &= \texttt{holdsat(f, I)} \\ Trans(\texttt{violations}) &= \texttt{occurred(viol(e, I))} \end{aligned}$$

For every parameter `p`, the `show` expression translation takes the form:

$$Trans(\texttt{show p}) = \begin{cases} \#\texttt{hide.} \\ \#\texttt{show}\ Trans(\texttt{p}). \end{cases}$$

In the next section we present some of the kind of reasoning tasks that we can perform using this query language.

6.3.4 Reasoning Tasks and Guidelines for Querying

The aim of our security requirements/policy analysis framework is to provide the system designer with the means of checking that the system model satisfies a set of desired security properties. The designer might also wish to use the analysis process to obtain information regarding the effect of non-compliance to a set of security requirements on the state of the system. These objectives suggest that, in broad terms, our framework must support three types of query namely: *validation*, *verification*, and *review* queries. Having given a description

of *pyisntql*, we illustrate here how it can be used for these reasoning tasks through query specifications. We categorise review tasks into two, namely: *prediction* and *post-diction* (Sergot, 2005) and are briefly described as follows:

Consider a transition system consisting of a set of all possible states $\mathcal{S}$, a set of all possible events/actions $\mathcal{A}$, a sequence of events/actions $A = a_1, \ldots, a_n$, and an initial state:

Prediction: Given that the transition system is in state $s \in \mathcal{S}$ and a sequence of events/actions $A = a_1, \ldots, a_n$ have occurred, the prediction task (s, A) is the problem of deciding the set of states $\{S' \subseteq \mathcal{S}\}$ which may result from the transition system. This can be expressed in *pyinstql* as:

```
condition X: A after(1) s;
constraint not X;
```

This query limits traces to those in which at some point s holds after which the events of A occur in sequence. The answer sets that satisfy this query will then contain the states $\{S' \subseteq \mathcal{S}\}$.

Post-diction: This is the converse of prediction such that if the system is in state s' and the events/actions $A = a_1, \ldots, a_n$ have occurred, then post-diction problem (A, s') is the task of deciding the set of states $\{S \subseteq \mathcal{S}\}$ that could have held before A. This implies that s' is required to hold in the next time instant following the final event of A. Post-diction can be expressed in *pyinstql* as:

```
condition X: s' after(1) A;
constraint not X;
```

Table 6.4 presents a guide on the types of query that may be used in investigating security properties from the model. In the next section we are presenting some concrete examples of how the *pyisntql* generates a query file from query statements.

Type of Query		Aim of Query	Example of Query
Validation		This query is for validating the model. Events need to occur in the sequence specified and all events and fluents in the specification must be represented in the answer set. This is achieved using `observe` *eventslist* in the query where *eventslist* is a list of events in the order required for the investigation.	`observe login(sHCP), logindl, selectAction(sHCP), makeRef(sHCP), checkRef(cHCP), checkRefdl, signRef(sys), encryptRef(sys), sendRef(sHCP), receiveRefwP(rHCP), enforcePolicy(sys); show events, states;`
Verification		This query verifies that the model satisfies the security properties being investigated. This is achieved by formulating the security property in form of a query.	The requirement: "The head shall not be in possession of the paper after sending it to the exam office" is expressible as: `condition compromised(hasPaper(head)): holds(hasPaper(head)) after happens(sendPaper(head,eo)); show compromised(hasPaper(head));`
Review	Prediction	This query enables the designer to investigate what the state of the system would be after a specified event has occurred	`condition systemState: holds(F) after happens(sendPaper(head)); constraint not systemState;`
	Post-diction	Post-diction queries enable the verification of how a particular state of the system came about	`condition transitionEvents: happens(E) after holds(hasPaper(head)); constraint not transitionEvents;`

Table 6.4: Summary of Query types

6.3.5 Example Queries

We illustrate how we may use *pyinstql* to form queries by recalling some queries used in the $MakeReferral$ scenario presented in Section 5.4.

We start with what we consider a fundamental task of validating our model. We

```
observed(login(sHCP), 0).
observed(logindl, 1).
observed(selectAction(sHCP), 2).
observed(makeRef(sHCP), 3).
observed(checkRef(cHCP), 4).
observed(checkRefdl, 5).
observed(signRef(sys), 6).
observed(encryptRef(sys), 7).
observed(sendRef(sHCP), 8).
observed(receiveRefwP(rHCP), 9).
observed(enforcePolicy(sys), 10).
```

Figure 6-6: Query for observed events

do this by querying for traces that are generated as a result of the exogenous (observable) events occurring in an order that conforms to the original work flow. The trace can be filtered by defining each exogenous event expected in the resulting answer set using the fact `observed(event,instant)` in the query specification. For example, following from the model of the misuse case presented in Section 5.3.2, the query in Figure 6-6 specifies the sequence of exogenous events that we are interested in. In *pyinstql* this query is expressed as:

```
observe login(sHCP), logindl, selectAction(sHCP), makeRef(sHCP),
       checkRef(cHCP), checkRefdl, signRef(sys), encryptRef(sys),
       sendRef(sHCP), receiveRefwP(rHCP), enforcePolicy(sys);
```

Notice that the sequence of events are expressed as a list separated by comma in an order that the events are expected to occur. Each event is then translated into equivalent fact in ASP as shown in Figure 6-6. The result of this query would give so much information than may be required. We therefore use the `show` expression to filter the result specific information we are verifying. For example, we may want to verify the correctness of sequence of events. We do this in *pyinstql* using the statement:

```
show events;
```

which gets translated in the query file as:

```
#hide.
#show occurred(E,I).
```

```
Answer: 1
occurred(makeRef(sHCP),3) occurred(receiveRefwP(rHCP),9) occurred(checkRefdl,5)
occurred(sendRef(sHCP),8) occurred(enforcePolicy(sys),10)
occurred(selectAction(sHCP),2) occurred(encryptRef(sys),7)
occurred(checkRef(cHCP),4) occurred(logindl,1) occurred(login(sHCP),0)
occurred(signRef(sys),6) occurred(loginChal(sys,sHCP),i02)
occurred(ilogin(sHCP),0) occurred(ilogindl,1) occurred(iselectAction(sHCP),2)
occurred(imakeRef(sHCP),3) occurred(icheckRef(cHCP),4) occurred(icheckRefdl,5)
occurred(isignRef(sys),6) occurred(iencryptRef(sys),7) occurred(isendRef(sHCP),8)
occurred(ireceiveRefwP(rHCP),9) occurred(ienforcePolicy(sys),10)
SATISFIABLE

Models : 1
Time : 0.000
  Prepare : 0.000
  Prepro. : 0.000
  Solving : 0.000
```

Figure 6-7: The result of model verification

This query simply hides all other results and only shows events occurrences and the time instant in which each event occurred. This is a very useful query which we shall be using to filter our answer sets to specific target results we are interested in seeing.

With these query specifications now, we can validate our model for correctness and completeness. Correctness requires that events and fluents occur in the order of sequence specified while completeness requires that all events and fluents are represented in the answer set. It is easy to validate for correctness because the result is expected to show an answer set that does not contain violations. Figure 6-7 presents the result of the validation query in Figure 5-11 on page 122 for events occurrences. This result validates the computational model for completeness and correctness since there is no violation in the result and all observed events specified in the query appear in the resulting answer set. The term `SATISFIABLE` in the result also means that there are models which satisfy the query.

It is important to note here that we can only validate the computational model as specified by the institution designer. Whether the institutional model in InstAL correctly captures the target system or work flow or not is beyond the scope of the work presented in this thesis. Therefore our validation here is subject to the correctness of the institutional model which yields the computational model we

are using for analysis as is the case with any abstraction.

Once we have validated the computational model, we can now use it to perform other analysis through the various reasoning tasks discussed in Section 6.3.4. This translator was used to generate the queries presented in Section 5.4.

6.4 *pyviz* - The answer set visualiser

Another issue with the use of the answer set program has to do with the nature of the answer sets produced. The answer sets produced from the output of the *clingo* solver for a translated institution program and query are expressed as a set of unordered atoms. It is difficult to see exactly what the trace and corresponding model described by the answer set means when examined in their original form. This is especially true in the case when many fluent values or generated events are used (Cliffe, 2007). In order to assist a designer in understanding the output of the answer set solver we have used a simple tool that allows the answer sets to be parsed and then displayed in a format that is easily readable and understood by users.

The trace visualiser tool *pyviz* takes a set of answer sets written in the output format of *clingo* and produces a graphical visualisation of the traces and models described by those answer sets. For each answer set specified in the input, the visualiser extracts the sequence of exogenous events represented by `observed(E,T)` atoms in the answer set that represent the trace, the set of generated events represented by `occurred(E, T)` derived for this trace and the corresponding fluent values for each state in the trace. These are then displayed as a sequence allowing a designer to see what has happened in a given trace.

We illustrate this with some examples:

The following shows an example of an answer set This is visualised as shown in Figure 6-9 on page 162. This looks like a state transition diagram with annotations on both the arcs and the nodes. The nodes S_i represent the states in the institutions transitions marked by the time instants i in which events occurred or fluents happened. The arrow lines represent transitions from one state or time instant to the other as events occur. The events which trigger the transitions are

```
Answer: 1
occurred(dointegrity(sHCP),4) occurred(sendRef(sHCP),3)
    occurred(makeRef(sHCP),2) occurred(selectAction(sHCP),1)
    occurred(login(sHCP),0) holdsat(live(patientRef),0)
    holdsat(perm(ilogin(sHCP)),0)
    holdsat(pow(patientRef,ilogin(sHCP)),0)
    holdsat(perm(login(sHCP)),0) holdsat(perm(login(sHCP)),1)
    holdsat(pow(patientRef,ilogin(sHCP)),1)
    holdsat(perm(ilogin(sHCP)),1) holdsat(live(patientRef),1)
    holdsat(live(patientRef),2) holdsat(perm(ilogin(sHCP)),2)
    holdsat(pow(patientRef,ilogin(sHCP)),2)
    holdsat(perm(login(sHCP)),2) holdsat(perm(login(sHCP)),3)
    holdsat(pow(patientRef,ilogin(sHCP)),3)
    holdsat(perm(ilogin(sHCP)),3) holdsat(live(patientRef),3)
    holdsat(live(patientRef),4) holdsat(perm(ilogin(sHCP)),4)
    holdsat(pow(patientRef,ilogin(sHCP)),4)
    holdsat(perm(login(sHCP)),4) holdsat(perm(login(sHCP)),5)
    holdsat(pow(patientRef,ilogin(sHCP)),5)
    holdsat(perm(ilogin(sHCP)),5) holdsat(live(patientRef),5)
    holdsat(perm(makeRef(sHCP)),4) holdsat(perm(dointegrity(sHCP)),4)
    holdsat(perm(makeRef(sHCP)),3) occurred(viol(sendRef(sHCP)),3)
    holdsat(perm(dointegrity(sHCP)),3) holdsat(perm(makeRef(sHCP)),2)
    holdsat(perm(selectAction(sHCP)),2)
    holdsat(perm(selectAction(sHCP)),1) occurred(ilogin(sHCP),0)
    holdsat(perm(iselectAction(sHCP)),1)
    holdsat(perm(iselectAction(sHCP)),2)
    holdsat(pow(patientRef,iselectAction(sHCP)),1)
    holdsat(pow(patientRef,iselectAction(sHCP)),2)
    occurred(iselectAction(sHCP),1) holdsat(perm(imakeRef(sHCP)),2)
    holdsat(perm(imakeRef(sHCP)),3) holdsat(perm(imakeRef(sHCP)),4)
    holdsat(pow(patientRef,imakeRef(sHCP)),2)
    holdsat(pow(patientRef,imakeRef(sHCP)),3)
    holdsat(pow(patientRef,imakeRef(sHCP)),4)
    occurred(imakeRef(sHCP),2) holdsat(perm(idointegrity(sHCP)),3)
    holdsat(perm(idointegrity(sHCP)),4)
    holdsat(perm(idointegrity(sHCP)),5)
    holdsat(pow(patientRef,idointegrity(sHCP)),3)
    holdsat(pow(patientRef,idointegrity(sHCP)),4)
    holdsat(pow(patientRef,idointegrity(sHCP)),5)
    holdsat(perm(dointegrity(sHCP)),5) occurred(idointegrity(sHCP),4)
    holdsat(perm(isendRef(sHCP)),5)
    holdsat(pow(patientRef,isendRef(sHCP)),5)
    holdsat(perm(sendRef(sHCP)),5)
SATISFIABLE
```

Figure 6-8: A sample answer set

displayed on top of the arrow lines. Institutional fluents that hold true at each state of the institution are displayed under each state (node) respectively. For in-

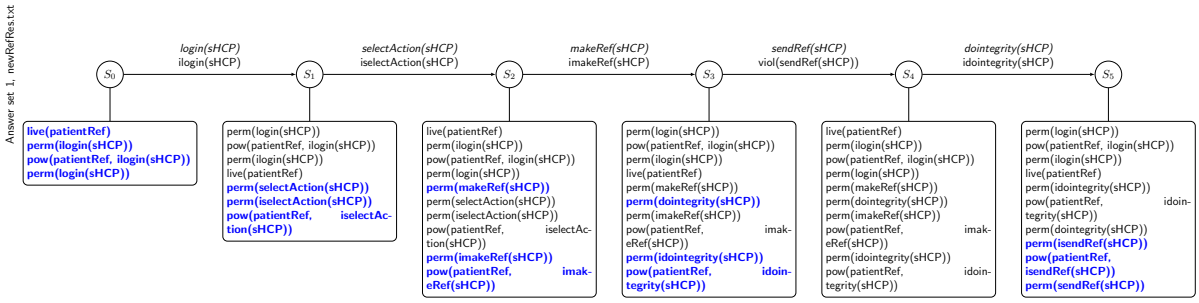


Figure 6-9: Answer set visualization.

stance at the initial state S_0 the exogenous event `login(sHCP)` has been granted permission. Also the corresponding institutional actions are both empowered and permitted at this time instant and state.

It is easy to follow through the transition in terms of how fluents evolve as the institution transits from one state to the other with new fluents added to each state indicated by bold characters. For example as the institution transits from state S_0 to state S_1 as a result of the institutional action `ilogin(sHCP)`, this results in the addition of the institutional fluents permitting the exogenous event `selectAction(sHCP)` and also empowering and permitting the institutional action `iselectAction(sHCP)`. This ability to be able see easily the effect of events on the state of the institution would be very helpful in spotting out possible vulnerability spots in the system interaction. This will then enhance decision on how security solutions may be applied to mitigate such security vulnerabilities.

Scalability of the visualiser

The sources of complexity for the visualiser are the number of instants, number of events, and the number of fluents in an answer set passed to the visualiser. Since the tool visualises all these components, it will certainly produce a very large diagram. For instance Figure 6-9 which visualises the answer set in Figure 6-8 illustrates how complex it can get. However, in the query based approach we present in this work, it is not likely that answer sets would contain a large number of these components, else the purpose of analysis using this approach would be defeated. The power in the use of the flags `#hide` and `#show`, introduced in Section 6.3.1 on page 149, in a query is that an answer set can be filtered to

produce traces that satisfy the query. The visualiser can also lay traces out in rows of a given length and can selectively display particular states required, and this ability helps with the scalability of the visualiser. We note that even if the unfiltered answer set is visualised (in which case the visualisation would be large depending on the number of time instants), the benefits lie in the fact that it is very much easier to read the result in a visualised form because the reader is able to quickly see what state changes at each time instant and what events occurred in between time instants.

6.5 Summary of Chapter

In this chapter, we presented the additional tools we have used in order to make our approach usable with minimal technical knowledge of the underlying technologies. First is the XML2InstAL translation. This tool was non-existing so we had to develop it from scratch taking advantage of the fact that UML diagrams can be represented in XML form. With the help of an appropriate XML parser, we are able to build the translator which translates models built in UML diagrams to InstAL model. The major limitation to this tool is the fact the a particular UML diagram tool has to be used in order to effective use our translator. Since this seems to us like a tool which will greatly make the institutional action language InsAL more user friendly, we shall be working on making it compatible with other UML design tools in the future. We have also developed the semantics needed for us to translate the UML Activity Diagrams to InstAL specifications.

Secondly we presented the *pyinstql* translator which allows us to write query descriptions in a form of human language which is then translator to an equivalent query specification file in $AnsProlog$. Our translator allows us to generate what we consider a complete query specification file with facts, rules, and flags.

Lastly, the *pyviz* tool presents our resultant models (answer sets) in a way that is significantly easier to read. This means that users are able to make meaning out of the results generated easily than the original text presentation of the result.

In the next chapter we shall be using these tools as we present query-based analysis of some security properties using our approach.

Chapter 7

Query-based Verification of Security Properties

7.1 Introduction

In the previous chapters we presented the institutional framework, action language InstAL and the *AnsProlog* program in which the computational model is expressed and made available for analysis through the use of answer set solvers. We have also presented how these tools may be used in the general sense of institutional modelling and specifically in the security domain for analysis of security requirements.

In this chapter we are going to be focusing mainly on performing analysis within the security domain.

7.2 Confidentiality scenario - Examination Paper process

Confidentiality scenarios are concerned with all security incidents and vector attacks that could be exploited to gain access to internal information and disclosing it to unauthorized people. We present a scenario involving the way examination papers are prepared and produced to be taken by students in a school. In this scenario, the asset to be secured is the examination question paper which needs to be protected from unauthorised actors. The risk involved is that the examination would loose its creditability if the question paper is exposed to students

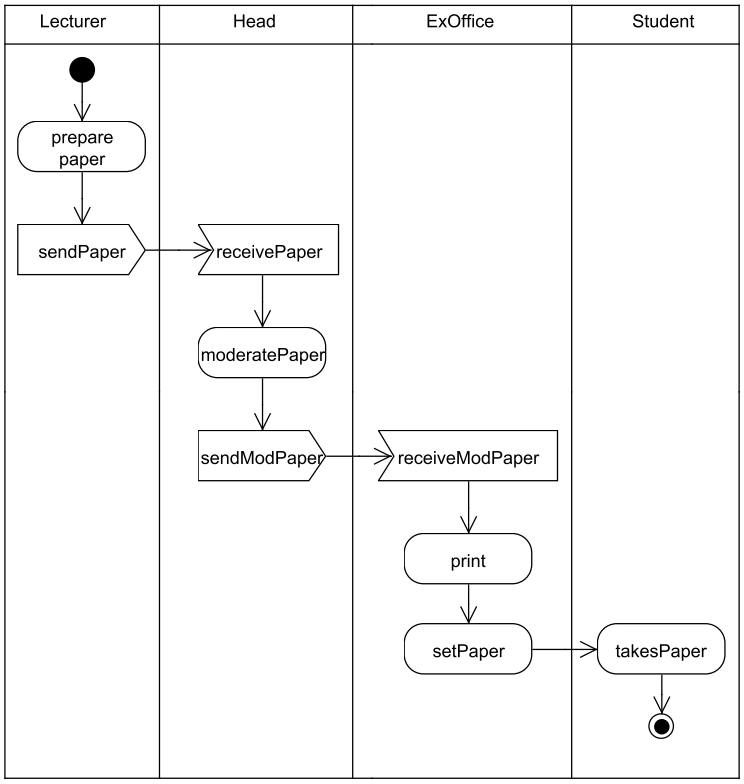


Figure 7-1: Examination Paper Process

before they are due to take the examination. Therefore the primary security goal in this scenario is *confidentiality* of the examination paper.

The model represents a segment of the process of preparing examination papers presented in the activity diagram (Figure 7-1). The model consists of the roles: Lecturer, head of department (Head), examination officer (ExOffice), and Student. The resource here is the examination question paper (Paper). The process starts with the lecturer preparing the question paper and sending it to the head of department (Head) who handles the moderation of the paper (this process is not included in our scenario). Assuming the paper has been moderated, the head sends the moderated paper to the examinations office which handles the printing of the paper and setting it to be taken by the student. The process ends with the student taking the paper.

In order to ensure that the confidentiality of the paper is preserved, the following security requirements are set:

R1: The paper shall not be disclosed to any other person before it is taken by the student

R2: The lecturer shall not be in possession of the paper after sending it to the head

R3: The head shall not be in possession of the paper after sending it to the exam office

With these candidate security requirements, it means that violation of any of these requirements would put the exam paper at risk and the paper would be considered insecure and unfit to be administered to the student. We shall be analysing these requirements using our framework for possible sources of confidentiality breaches.

Requirement **R1** will require us to introduce another actor `misuser` such that any interaction with this actor would be suspicious. For instance, if any of the actors performs a `"send"` action to `misuser`, this would be expected to trigger a violation. The requirements **R2** and **R3** would demand actions that would dispossess the actors of the assets at the specified time instants. An action such a delete action would be used to represent this.

```
1  Lecturer: lecturer
2  Student: student
3  ExOffice: eo
4  Head: hod
5  Paper: paper
6  Actors: hod misuser eo student lecturer
```

Figure 7-2: The Domain file Examination Paper Process Model.

We modelled the scenario in InstAL as shown in Figure 7-3 with the associated domain file in Figure 7-2. In modelling the scenario, we initially permit the lecturer to prepare the paper `perm(prepare(lecturer, paper))`. This assumes the state of the institution from its start. Also the corresponding institutional actions are empowered and permitted. Other exogenous events are permitted at various instants as the institution evolves.

```
institution examPaper;

type Lecturer;
type Student;
type ExOffice;
type Head;
type Paper;
type Actors;

% Exogenous events
exogenous event prepare(Lecturer,Paper);
exogenous event sendPaper(Lecturer,Actors);
exogenous event sendModPaper(Head,Actors);
exogenous event print(ExOffice,Paper);
exogenous event setPaper(ExOffice,Actors);
exogenous event takes(Student,Paper);

% Institutional events
inst event iprepare(Lecturer,Paper);
inst event isendPaper(Lecturer,Actors);
inst event isendModPaper(Head,Actors);
inst event iprint(ExOffice,Paper);
inst event isetPaper(ExOffice,Actors);
inst event itakes(Student,Paper);

%fluents
fluent hasPaper(Actors);
fluent hasModPaper(Actors);
fluent setToTake(Student,Paper);
fluent hasTaken(Student,Paper);
fluent hasPrintedPaper(Actors);

% conventional generation
prepare(Lecturer,Paper) generates iprepare(Lecturer,Paper);
sendPaper(Lecturer,Actors) generates isendPaper(Lecturer,Actors);
sendModPaper(Head,Actors) generates isendModPaper(Head,Actors);
print(ExOffice,Paper) generates iprint(ExOffice,Paper);
setPaper(ExOffice,Actors) generates isetPaper(ExOffice,Actors);
takes(Student,Paper) generates itakes(Student,Paper);

%% creation of patient referral
iprepare(Lecturer,Paper) initiates perm(sendPaper(Lecturer,Actors)),
    perm(isendPaper(Lecturer,Actors)), pow(isendPaper(Lecturer,Actors)),
    hasPaper(Lecturer);

iprepare(Lecturer,Paper) terminates perm(prepare(Lecturer,Paper)),
    perm(iprepare(Lecturer,Paper)), pow(iprepare(Lecturer,Paper));

isendPaper(Lecturer,Head) initiates perm(sendModPaper(Head,Actors)),
    pow(isendModPaper(Head,Actors)), perm(isendModPaper(Head,Actors)),
    hasPaper(Actors);

isendPaper(Lecturer,Head) terminates perm(sendPaper(Lecturer,Actors)),
    perm(isendPaper(Lecturer,Actors)), pow(isendPaper(Lecturer,Actors)),
    hasPaper(Lecturer);

isendModPaper(Head,ExOffice) initiates perm(print(ExOffice,Paper)),
    pow(iprint(ExOffice,Paper)), perm(iprint(ExOffice,Paper)),
    hasModPaper(Actors);

iprint(ExOffice,Paper) initiates perm(setPaper(ExOffice,Actors)),
    pow(isetPaper(ExOffice,Actors)), perm(isetPaper(ExOffice,Actors)),
    setToTake(Student,Paper), hasPrintedPaper(Actors);

iprint(ExOffice,Paper) terminates hasModPaper(ExOffice);

isendModPaper(Head,ExOffice) terminates perm(isendModPaper(Head,Actors)),
    pow(isendModPaper(Head,Actors)), perm(sendModPaper(Head,Actors)),
    hasModPaper(Head);

isetPaper(ExOffice,Student) initiates perm(takes(Student,Paper)),
    pow(itakes(Student,Paper)), perm(itakes(Student,Paper));

isetPaper(ExOffice,Student) terminates perm(setPaper(ExOffice,Actors)),
    pow(isetPaper(ExOffice,Actors)), perm(isetPaper(ExOffice,Actors)),
    hasModPaper(Actors);

itakes(Student,Paper) initiates hasTaken(Student,Paper);

%%initial states
initially
  perm(prepare(lecturer,paper)), pow(iprepare(lecturer,paper)),
      perm(iprepare(lecturer,paper));
```

Figure 7-3: The Examination Paper Process Model in InstAL.

Model Validation: The first thing we need to consider is whether our computational model represents the system we are analysing. In order to do this validation, we run the model with the following query:

```
observe prepare(lecturer,paper),sendPaper(lecturer,hod),
sendModPaper(hod,eo), print(eo,paper),
setPaper(eo,student), takes(student,paper);

show states;
```

The query specifies the order in which exogenous (observed) events ought to occur. It also specifies that we want to see specifically the state of the process at various time instants as the institution evolves with occurrences of events. What we expect to see is a single answer set that shows the occurrence of the events stated in the query without any violation event. With respect to ordering of events being tested here, violation events will result with the occurrence of an event at the time instant that it is not permitted to occur. This will be interpreted to mean existence of a flaw in the computational model. Also the fluents declared in lines 27–31 of the model in Figure 7-3 are expected to hold at certain time instants, showing the state of the system in terms of the progression of the work flow at those time instants. Another important consideration to be noted is the number of time steps required to run the model in order to produce the required model. This is determined by the number of exogenous events specified in the query as follows:

$$Timesteps = numberofobservedevents + 1$$

The implication of running the model for a time instant less than the required number of time instants is that not all the observed events specified would be processed, as such the answer set would be incomplete. On the other hand if the time instant is greater than the required number, the process gets repeated, thereby producing longer traces and answer sets. Violations would usually occur in such cases due to the fact that events would be occurring at more than one time instant.

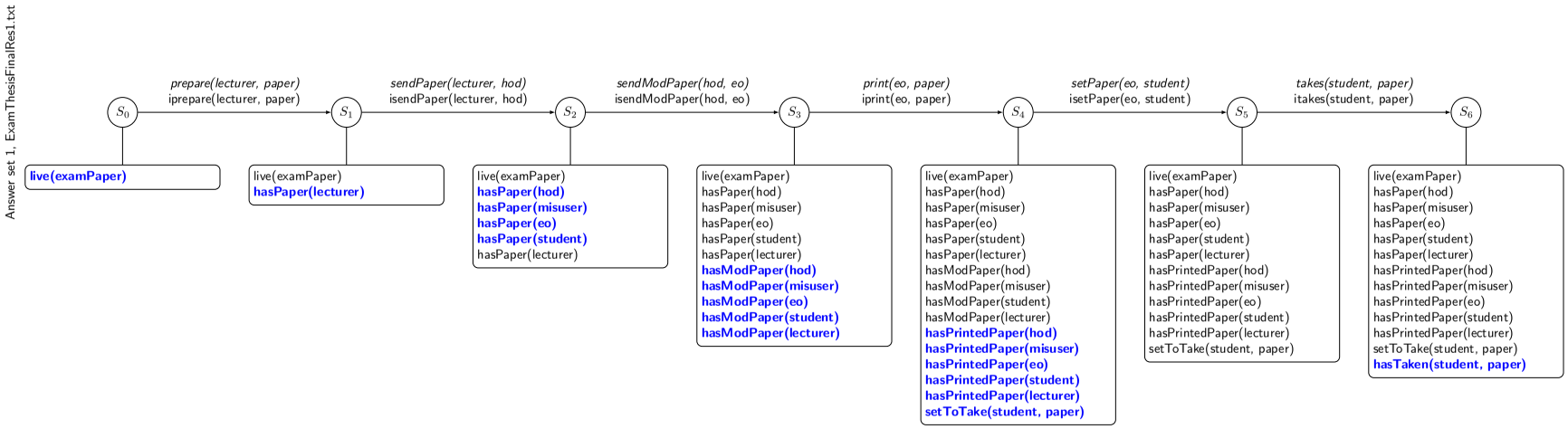


Figure 7-4: Model validation

Figure 7-4 shows the visualisation of the produced answer set. The time instants are represented by the nodes S_n, the arrows show the direction of transition of the model from one state to the other. The occurred events (exogenous, institutional, and violation) which trigger the transition from one state to the other appear as labels on the appropriate transition arrows. The state of the system at each time instant (generally expressed as fluents) appear under the appropriate state nodes. Fluents are added and removed according to the model design as the institution evolves. Our visualisation does not show the removed fluents since we do not consider them as useful information. However, it is important to be able to see easily what new fluents are added to the states as the institution evolves. These are coloured in blue to differentiate them from the existing fluents which still hold at previous instants and still hold at the current instant.

Running our model for seven(7) time steps (since we have six observed events) produces the visualised answer set which shows that there is no occurrence of violation events and it contains all the specified observed events. The answer set also shows that the fluents hold at time instants as expected from the model. For instance,we expect that fluent `hasPaper(lecturer)` to hold at the next time instant after the event `iprepare(lecturer, paper)` has occurred (Figure 7-3 on page 167, line 42). The visualised answer set shows that this fluent holds as expected. Similarly, line 50 of the model specifies that the fluent `hasModPaper(Actors)` holds at the next time instant after the event `isendModPaper(hod,eo)` (following definitions of `Head` and `ExOffice` in the domain file - Figure 7-2) must have occurred. This is correctly shown in the answer set visualisation with the fluent holding for each of the actors defined in the domain file. With the model validated, we can now use it to analyse the requirements specified earlier.

Analysis of the Model: For a security requirements designer who has the goal of preserving the confidentiality of the target assets, the interest would be in knowing the points at which the assets would be vulnerable to security breaches. This information would be fundamental for designing the appropriate confidentiality security requirement that would mitigate the vulnerability at those points. Therefore the question that would be of interest here is:

What would be the confidentiality state of the asset Y associated with event X after the event X has occurred?

This is clearly a *prediction* problem. Here we would like to see the state of the system

after a certain event has occurred. Using our scenario for example, the occurrence of the event `sendPaper(lecturer,hod)` at time instant `1` means that institutionally, the `lecturer` is permitted to send the prepared paper to the `hod` from that time instant. However looking at the problem from human factor perspective, since the `lecturer` has possession of the paper and the power[1] to send out the paper, it means that the `lecturer` can also send the paper to other actors other than the `hod` who is the only recipient permitted by the institution, either intentionally or otherwise. Whatever the case, the action would compromise the confidentiality of the paper. We would verify this by querying for the state of the institution from the time instant after the `sendPaper(lecturer,hod)` event has occurred.

Using the following query:

```
condition compromised(hasPaper(X)): happens(sendPaper(lecturer,hod))
after holds(hasPaper(X)) and X not hod;

show compromised(hasPaper(X));
```

This considers the possession of the paper by any actor `X` at any time instant `J` to be a compromised where `J` is a time instant after instant `I` and `X` is not `hod`. The resultant answer set is visualised as seen in Figure 7-5.

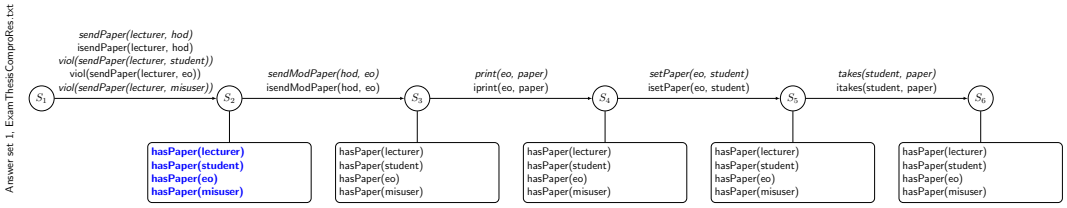


Figure 7-5: Verification

Explaining the result:

The result visualised (Figure 7-5) shows that from the time instant 2 after the paper has been sent to the `hod`, other actors could be in possession of the paper. This is possible from the fact that the paper could be leaked to these other actors from that time instant. The result also shows that this state of the paper could persist to the end of the process. What this means is that there is a potential for the confidentiality of

[1] All exogenous events are empowered by default since they are outside the institution, hence cannot be controlled by the institution.

the paper to be compromised from the moment the `lecturer` sends the paper to the `hod`.

The implication of this information in terms of designing the security requirement is that there would be a need for requirements that restrict the `lecturer` from sending the paper to any other actor other than the intended recipient. This may also include requirements that have to do with the form of storage (digital or physical).

The result shown in Figure 7-4 on page 169 indicates that similar vulnerabilities could arise at other time instants too as the paper is being processed. These are useful information for designing the appropriate requirements to mitigate these vulnerabilities.

7.3 Integrity scenario - Patient referral management

We revisit the patient referral management scenario from the iTrust Medical Records System presented in Section 5.3. This scenario involves actors interacting with a vital information - the patient's referral document. The integrity of this document is of utmost importance than confidentiality, although confidentiality could also be seen as a security goal for this asset. Alteration of the referral details for instance could lead to either delay in progression of the treatment or administration of a different line of treatment other than what it ought to be. Either of this could be critical to the patient. It is therefore important to make the preservation of integrity a priority here.

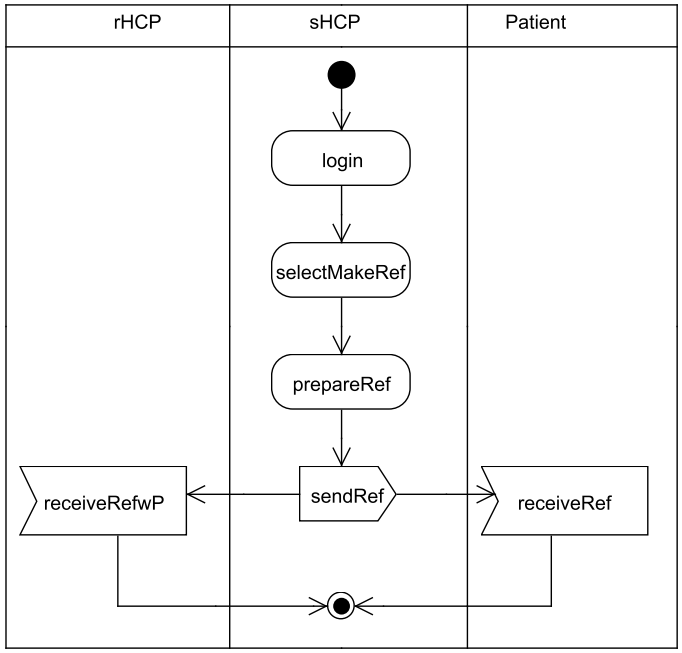


Figure 7-6: The Basic Activities of the 'Make Referral' Scenario

The basic activity flow for the *Make Referral* scenario is presented in Figure 7-6. The corresponding instAL model is presented in Figure 7-7.

The model captures the scenario in its simplest form. This is intended to make it simple enough to understand how we use the tool to analyse the integrity requirement for the target asset. As we did in the previous section, the model validates the UML model to the model shown in Figure 7-8 on page 175. As the process evolves, new fluents (in blue colour) are added to the states of the institution. Notice that even though the events (shown on the transition lines) do not have any violation events occurring, the fluents in the states from S_1 show permissions for events triggered by both `sHCP` and `rHCP`. Since the permissions are not terminated (a situation in which no controls are applied), they persist till the end of the process.

This result clearly shows the various possible points at which the process and the referral document could be vulnerable to integrity breaches. From this scenario, valid referral document can only be prepared and sent by `sHCP`, therefore any other actor performing any of these tasks would be considered a misuse and hence pose a risk

```
1  institution makeRef;
2
3  %% types declaration with domain file specification as comments
4  type HCP; %sHCP rHCP
5  type Patient; %patient
6  type System; %sys
7
8  % Exogenous events
9  exogenous event login(HCP,System);
10 exogenous event selectMakeRef(HCP);
11 exogenous event prepareRef(HCP);
12 exogenous event sendRef(HCP);
13
14 % Institutional events
15 inst event ilogin(HCP,System);
16 inst event iselectMakeRef(HCP);
17 inst event iprepareRef(HCP);
18 inst event isendRef(HCP);
19
20 % fluents
21 fluent hasRef(Patient);
22 fluent hasRefwP(HCP);
23
24 % conventional generation
25 login(HCP,System) generates ilogin(HCP,System);
26 selectMakeRef(HCP) generates iselectMakeRef(HCP);
27 prepareRef(HCP) generates iprepareRef(HCP);
28 sendRef(HCP) generates isendRef(HCP);
29
30 ilogin(HCP,sys) initiates perm(selectMakeRef(HCP)), pow(iselectMakeRef(HCP)),
       perm(iselectMakeRef(HCP));
31
32 iselectMakeRef(HCP) initiates perm(prepareRef(HCP)), pow(iprepareRef(HCP)),
       perm(iprepareRef(HCP));
33
34 iprepareRef(HCP) initiates perm(sendRef(HCP)), pow(isendRef(HCP)),
35     perm(isendRef(HCP));
36
37 isendRef(HCP) initiates hasRef(Patient),hasRefwP(HCP);
38
39 %initial states
40 initially
41   perm(login(sHCP,sys)), pow(ilogin(sHCP,sys)), perm(ilogin(sHCP,sys));
```

Figure 7-7: The The Inst*AL* model for the *Make Referral* Scenario.

to the integrity of the process and the final referral document sent and received. We summarise the possible vulnerability points as follows;

- Assuming that `rHCP` got the login details to the system and successfully login, it means that from S_1 this actor can initiate the process of preparing the referral document. If this is allowed to happen, the integrity of the final referral document prepared and sent would be questionable. Therefore this is one point of mitigation that would be worth considering when designing the integrity requirement for the process. Such a mitigation can be to specify a requirement for a

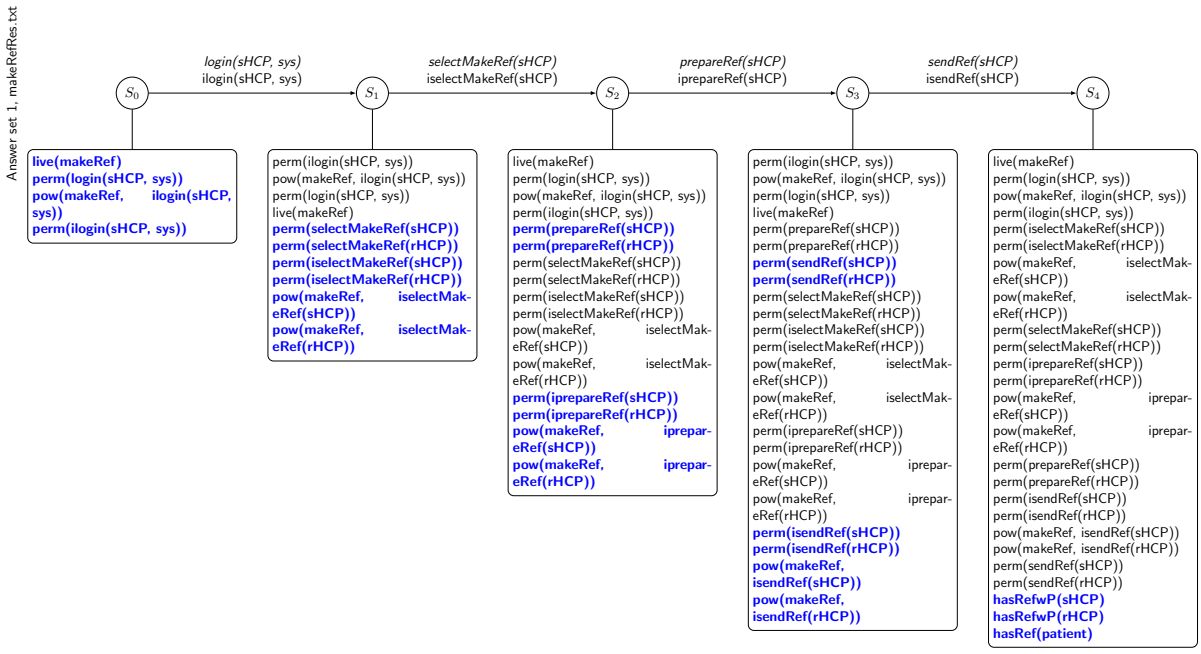


Figure 7-8: The 'Make Referral' Model Validation

counter check of the actor's identity using an appropriate mechanism such as a challenge-response mechanism.

- From state S_2 we can deduce a vulnerability that has to do with the preparation of the referral document. Assume that the `sHCP` started preparing the document but could not finish it for some reason and saved it. A misuser (`rHCP` in this case) who obtained access to the system could alter the details such that `sHCP` may not notice it. This also calls for consideration when designing the integrity requirements for the process.

- The vulnerability that we can deduce from state S_3 stems from the vulnerability in state S_2. A misuser who prepares a wrong referral document is able to send such to the recipients. Notice that the recipients are going to be receiving different versions of the same document. The `patient` receives an ordinary referral document, while `rHCP` receives the referral document which contains extra information such as the priority level of the referral. These extra information are also at risk of being altered and sent. There has to be a consideration of these possibilities when designing the integrity requirements.

7.4 Verification of Security Requirement Compliance

The formulation and specification of security requirements is usually done at different levels of abstraction and guided by the security objectives of the organisation. These are reflected in various control documentations such as standards, regulations, legislation, directives, and mission statements. These documents such as the UK Data Protection Act 1998, the USA Health Insurance Portability and Accountability Act (HIPAA) 1996, and the Sarbanes-Oxley Act (commonly named SOX), enacted in 2002 in the USA all describe mandates and requirements for the protection of vital information and resources in their various domains of application. In addition to these, the ISO/IEC 27001 Information Security Management defines the standards particularly for security management. These documents therefore form the basis on which organisations engineer security requirements. It is important to note that security requirements are high level specifications of the kinds of security expected in the organisation with respect to the organisation's assets and security goals of confidentiality, integrity, and availability without stating the mechanisms to bring about the expected security.

Since the focus of this thesis is not on the specification of security requirements but rather the analysis of the specified requirements, this section is dealing with the analysis of the security requirements particularly the verification of the requirements compliance. Verifying the compliance of systems and employees with security requirements is an important issue to be addressed if the security goals of the organisation are to be achieved.

Requirements generally consist of the following;

- triggering actor
- the keyword *shall*
- the action to be triggered
- conditions under which the actions can be triggered.

One form of expressing requirements is:

The **System** *shall* **[do something]** *[optional conditions]*

We shall be looking at analysing the compliance of security requirements expressed in this form. For example we use the following authentication requirement statement;

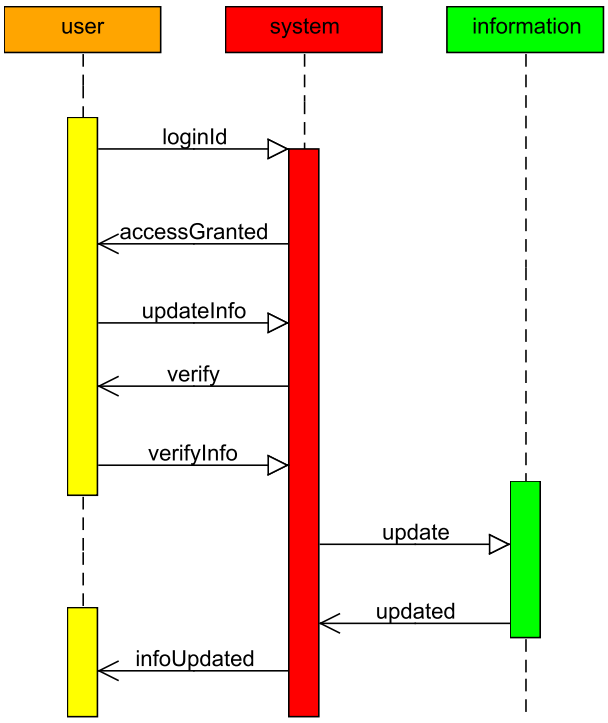


Figure 7-9: The requirement R_1

R_1. *The* **system** *shall* **verify** *the identity of all of its users* **before** *allowing them to update their user information.*

Applying the Institutional Framework

The implication of the requirement R_1 in terms of compliance is that the user can only perform an `update` action on their information subject to the actor `system` performing the action `verify` on the user's identity. The institutional framework allow us to express requirements like R_1 using the notion of *obligation* introduced in Section 3.6.1.2 on page 54. With this we may verify the compliance of this requirement by checking that in any process that requires compliance of this requirement, the obligations triggered by the requirements are fulfilled.

We present the requirement R_1 as a sequence of activities in Figure 7-9 which describe what may be seen as a compliant behaviour.

The requirement R_1 may be expressed as;

obl(verify, update, violation)

which means that whenever the obligation holds, the event `verify` must occur before the event `update` occurs, else the `violation` event would be triggered.

We see the problem of verifying compliance of the system to the requirement as a reasoning problem in the action theory. We therefore use the InstAL action language to model the scenario presented in Figure 7-9. This will generate the computational model for us in ASP from which we can do verification by querying the answer sets for compliance or lack of it.

In providing the specification of the model in InstAL, we define the descriptions presented in Figure 7-10. We define three types of interacting components `Agent`, `System`, and `Info`. How these components relate with each other is defined by three exogenous (observable) events `login(Agent,System)`, `verify(System, Agent)`, and `update(Agent, Info)` respectively. We also define a violation event

`violAuthentication(System)`

which is included in the definition of the compliance obligation and would be triggered whenever the obligation is not fulfilled. The obligation is defined as `obl(verify(System, Agent), update(Agent,Info), violAuthentication(System))`.

Model Validation: In order to validate the computational model, we shall check to see that if the events occur at the correct sequence as described in the sequence diagram, we should have only one answer set as our result. Also importantly, the result should not have any violations due to occurrence of non-permitted events, and there should not be any occurrence of the violation event as a result of the lack of fulfilment of the obligation.

To achieve this, we use the following query specification to query the answer set:

```
1  observed(login(employee,sys),0).
2  observed(verify(sys,employee),1).
3  observed(update(employee,infor),2).
4
5  #hide.
6  #show occurred(X,I).
7  #show holdsat(X,I).
```

This produces the result shown in Figure 7-11 on page 180:

```
1  institution authentiC;
2
3  type Agent;
4  type System;
5  type Info;
6
7  % exogenous events
8  exogenous event login(Agent,System);
9  exogenous event verify(System, Agent);
10 exogenous event update(Agent, Info);
11
12 % institutional events
13 inst event ilogin(Agent,System);
14 inst event iverify(System, Agent);
15 inst event iupdate(Agent, Info);
16
17 % violation event
18 violation event violAuthentication(System);
19
20 % fluents
21 fluent loggedIn(Agent,System);
22 fluent verified(System,Agent);
23 fluent updated(Agent,Info);
24
25 % obligation fluent
26 obligation fluent obl(verify(System, Agent), update(Agent,Info),
       violAuthentication(System));
27
28 % generates rules
29 login(Agent,System) generates ilogin(Agent,System);
30 verify(System, Agent) generates iverify(System, Agent);
31 update(Agent, Info) generates iupdate(Agent, Info);
32
33 %%% consequence rules
34 ilogin(Agent,System) initiates perm(verify(System, Agent)), pow(iverify(System,
       Agent)), perm(iverify(System, Agent)), obl(verify(System, Agent),
       update(Agent,Info), violAuthentication(System)), loggedIn(Agent,System);
35
36 iverify(System, Agent) initiates perm(update(Agent, Info)), perm(iupdate(Agent,
       Info)), pow(iupdate(Agent, Info)), verified(System,Agent);
37
38 iupdate(Agent, Info) initiates updated(Agent,Info);
39
40 %initially
41 initially perm(login(Agent,System));
42 initially perm(ilogin(Agent,System));
43 initially pow(ilogin(Agent,System));
```

Figure 7-10: InstAL model of the compliance scenario.

The resultant model (answer set) shows the three observable events occurring in the expected sequence without the occurrence of violations. Another thing we are interested in checking is the instances in which the fluents hold. A valid model would be one in which in addition to the events occurring in an order which does not result in violations occurring, every fluent should hold at the appropriate instants as defined. In Figure 7-11 we see that all the fluents defined hold at the right instants. For instance, at the instant

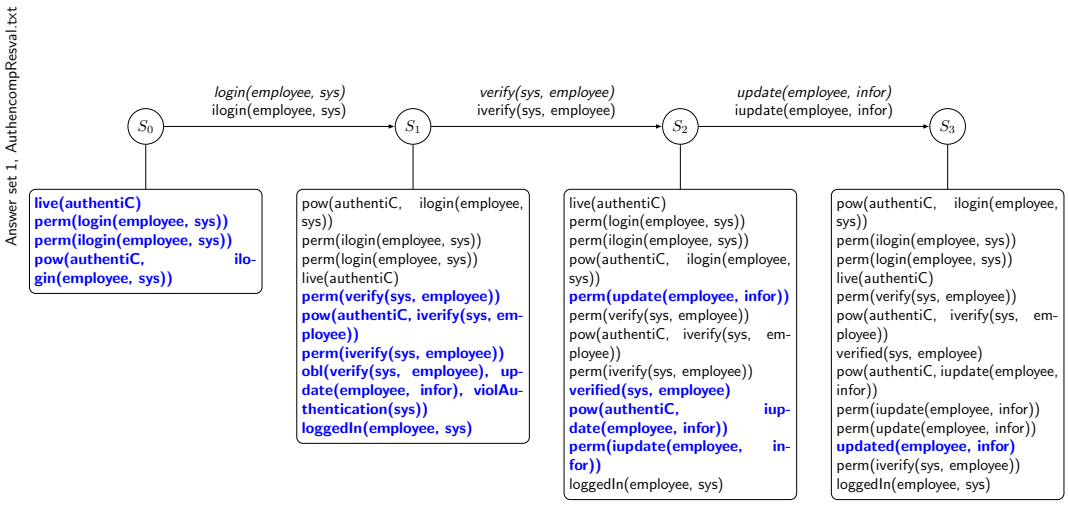


Figure 7-11: The result of computational model validation for R_1

defined by S_1, the obligation fluent holds as well as the fluent `loggedIn(employee,sys)` indicating that the employee has logged in to the system at this instant. Similar deductions can be made from the other time instants. These satisfy our validation criteria, and we can therefore begin to analyse the model for compliance issues.

Analysing Non-compliance: Having validated the model, we will be showing how non-compliance may affect the system. This analysis would give the requirements engineer insight into what non-compliance would look like and from there, give insight into how to better design the requirement.

First we would like to show a case of simple non-compliance: Assuming the requirement that user be verified before granting update permission is not complied with by ignoring it. In essence, the user logged in, and went straight to perform update, bypassing the verification procedure.

We check this by using the following query:

```
1  observe
       login(employee,sys),update(employee,infor);
2  show events, holds;
```

This query results in a number of answer sets presented in Figure 7-12 which reveal different information to the designer. For instance, Figure 7-12a shows an attempt to `update` information by `employee` after time instant S_1. This attempt flags the action as a violation `viol(update(employee, infor))`. At this period also, since the obligation

is not fulfilled, the violation event `violAuthentication(sys)` was also triggered. It can be observed also that the fluent associated with the update event did not hold as expected. However, since the employee is still logged in, another attempt at update after the time instant S_2 only flagged the event as a violation but no violation event is triggered here. The implication of this is that the employee could get away with this update activity if there is nothing in place to check the status of each event as it occurs in real time. One of the strategies organisations employ towards achieving compliance is the retrospective reporting where audits are conducted for after-the-fact detection by analysis of the log data whenever something goes wrong due to non-compliance or some other security breach.

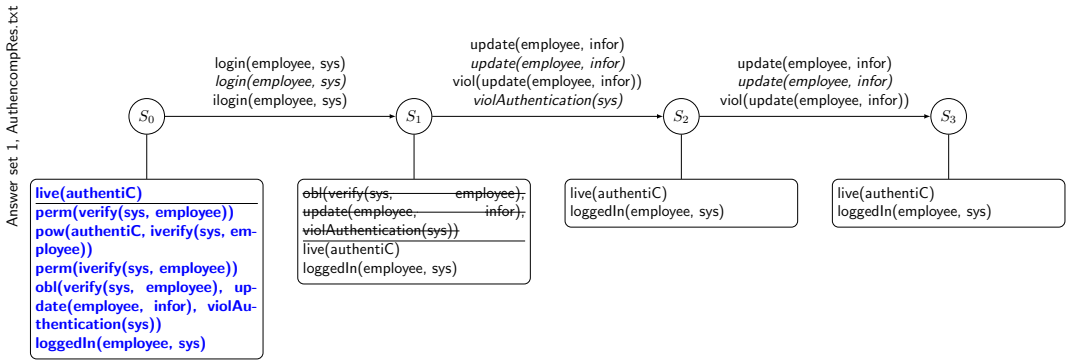
We can also observe from the results presented in Figure 7-12b that despite the fact that the update event flagged a violation and a violation event been triggered due to the non-fulfilment of the obligation at time instant S_1, the event `verify` still occurred after instant S_2 which results in the associated fluent `verified(sys,employee)` to hold at instant S_3. What this means is that there is the tendency for a cover-up event to occur such that the non-compliant behaviour may look like a compliant behaviour when the audit log is viewed without taking note of the instants at which the events occurred.

Lastly Figure 7-12c shows another possible behaviour of the system whenever there is non-compliance. This time, after the violation event `violAuthentication(sys)` has been triggered, the event `login(employee,sys)` could be initialised again, in which case, the obligation fluent is also initiated as seen at instant S_3. This attempts to repeat the process again. While this looks like a self-recovery behaviour by the system, if allowed this could keep the process going in a loop thereby resulting probably in a denial of service.

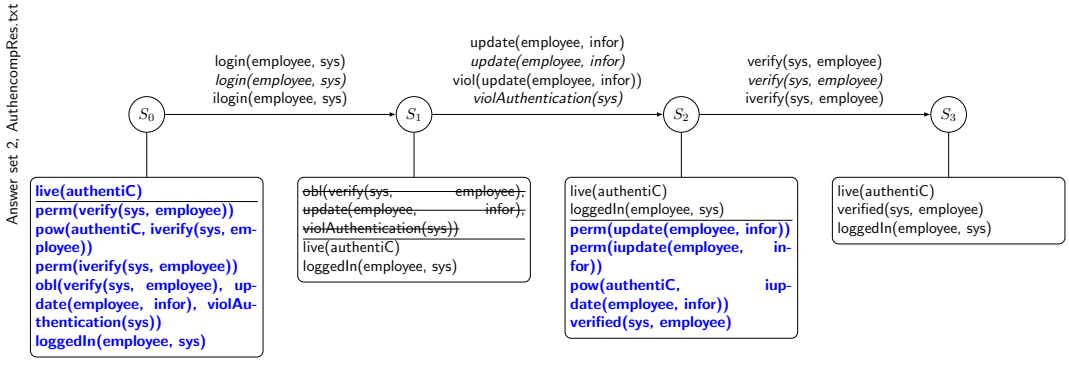
7.5 Summary of Chapter

In this chapter we presented illustrations on how the our computational approach can be used to verify security properties. Applying the technique of querying, prediction analysis can be performed on the computational model of a system such that points of security vulnerabilities can be easily seen. The information derived from the analysis will guide in the design of the appropriate security requirements that would mitigate the security threats.

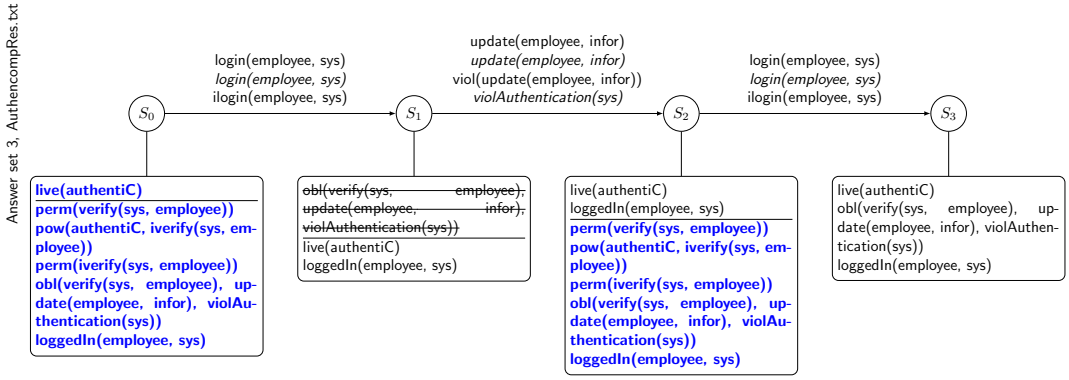
We applied our technique to investigating security properties using different scenarios



(a)



(b)



(c)

Figure 7-12: The effects of non-compliance

with different security goals, based on the target assets in the scenarios.

We have explained how our approach can be applied to analysing non-compliance of security requirement with an example. It shows the possible behaviours of the system whenever there is non-compliance. Being able to see these behaviours at design time using this static analysis would help in the design of appropriate mitigations to the revealed vulnerabilities.

Chapter 8

Summary and Conclusions

The drive for the security of information in organisations is far from being over. Despite increased investments in security through implementation of security solutions, security still remains a concern for organisations as reported by recent surveys on information security. These challenges are largely due to the human factor element which is regarded as the "weakest link" in the security chain. Much research in information security has seen security as a technical problem. However we approached the problem from the fact that organisations are socio-technical in nature, that is a system consisting of not only technological components but also human participants who interact with one another as they go about their duties. Hence in order to address the problem of security in organisations, the requirements to mitigate security vulnerabilities need to be analysed in view of the fact that human behaviours could also be a source of information security breach.

Having presented our research in the previous chapters, we now summarise our contributions in this chapter. We also examine possible extensions and future work from this point.

8.1 Summary of Contributions

The contributions of this dissertation are summarised as follows:

1. In Section 1.2 on page 11 we discussed the relationship between logic and security, particularly when considering the human factor in information security management. We therefore established that we can computationally reason about security requirements at design time using the institutional framework which allow

us to model actions in terms of permissions, obligations, and empowerment.

2. In Chapter 5 Section 5.2 on page 107, we present a computational approach for the analysis of security requirements using an institutional framework which is based on logic programming and inspired by deontic logic. We model scenarios in InstAL, an action language which abstracts from Answer Set Programming and in Section 5.4 on page 121 we demonstrate how reasoning can be carried out over the computational model by querying.

3. We develop a tool for generating InstAL specifications from UML Activity Diagram models, thereby extending the original InstAL development process. To achieve this we develop the semantics for translating UML Activity Diagrams to InstAL based on Petri Net workflow semantics (see Section 6.2.1 on page 131). In order to allow for expression of queries in a more natural language manner, we extend an existing query translation tool in Section 6.3 on page 148.

4. In Chapter 7 we demonstrate how our methodology which is based on logic and institutional frameworks can be applied to the security domain. We use a query based approach to analyse security properties of confidentiality (Section 7.2 on page 164) and integrity (Section 7.3 on page 172).

5. Managing information security requirements compliance is also an issue with security in organisations. Security solutions implemented can only be as successful as the degree of compliance that is achieved. In Section 7.4 on page 176 we demonstrate how security requirements compliance can be verified at design time using our framework.

8.2 Future work

We have focused on the verification of non-functional security requirements in this dissertation. There are a number of extensions and directions for future work:

8.2.1 Run-time verification of security requirements

So far we have done only static analysis of security requirements. A next step would be to analyse security requirements at run time. This would involve agent-based simulation (Macal and North, 2005) in which participants in an organisation would modelled as agents and the behaviours of these agents analysed in terms of the impact of their compliance or violations to the security requirements.

8.2.2 Secure process design

Several approaches have been proposed to improve compliance among employees. These include sanctions and rewards, awareness and training (Bulgurcu et al., 2008; Puhakainen and Siponen, 2010). These approaches try to tackle the compliance problem from either deterrence or detection and recovery approaches. However the challenge still remains that deterrence approaches have not significantly reduced non-compliance (Critchley, 2009). This is largely due to the fact that employees consider the level of convenience the security requirements afford them for compliance as they strive towards meeting their personal performance targets at work. We see an opportunity for application of our methodology here in influencing the business process design. The answer sets generated from the computational model of a scenario would suggest which of the models would provide less inconvenience towards compliance by the participants if implemented. The level of convenience can be measured by the amount of violations that occur, if we assume that the violations signify inconvenience.

8.2.3 Integration of Tools

We have developed tools in order to enhance the modelling, the analysis, and the understanding of the query results obtained. However, there is still limitation in the use of some of these tools. For instance the UML2Inst*AL* translator can only work on a specific UML design tool. This is due to the fact that the different UML tools available have a different way that they render the XML (or its likeness) representation which is used to generate the Inst*AL* specifications. There is the opportunity to extend this tool such that it should be able to work with a number of other UML tools. It is desirable that the tools be integrated into a common user interface to enhance their usability.

8.2.4 Security Requirements in complex systems

As the complexity in socio-technical systems increases, so also the challenge of information security. Systems composed of information, physical components, and human behaviours become more sophisticated as boundaries between organisation face dissolution due to outsourcing, proliferation of mobile devices and service composition. These trends lead to a significant increase in the number of possible interactions between the participants in such systems. From security point of view, developments such as working from home, bring-your-own-device (BYOD), and cloud computing result in

increasingly complicated information security problems (Pieters et al., 2013). Security problems such as propagation of access rights in complex attack scenarios is an issue that needs to be dealt with. Attacks which may include physical access and social engineering may be exploited at different vulnerability levels. An example is the road apple attack (Stasiukonis, 2006) scenario where an attacker leaves infected dongles around the organisation's premises. Access rights are propagated as an employee picks up an infected dongle and plugs it into the organisation's computer, malware will send out all the information it can find. Important questions that come with the possibilities for such multi-step attacks in increasingly complex systems include how to manage information security requirements in such complex situations, and how to check whether the security requirements are adequate. This is an area that we would like to explore with our methodology.

8.2.5 Conflicts between Organisational Norms and Information Security Requirements

Take the following scenario for instance: a department's security requirements include a policy that only authorised employees and visitors can enter the premises. A visitor would need to first get a visitor's access card at the reception. However, the automated entrance doors are on a time delay and stay open for a while after an authenticated person has been granted access and would allow non-registered persons to tailgate behind some one with an access card. The employees have been told not to let this happen, but it occurs regularly anyway. There is a conflict here between the policy, the ethical norm of not treating people as suspicious without a good reason, and the organisational norm of politeness to visitors. This problem has been identified and presented in Pieters and Coles-Kemp (2011) but there is no computational solution to this problem. This is an area that our work can be extended to analyse the relationship between the possibly conflicting organisational norms and security requirements at design time in order to guide the formulation of such security requirements.

8.3 Concluding Remarks

This thesis presents an approach to security requirements verification and validation that makes use of formal reasoning methods. We are motivated by the fact that the interaction of various participants in a system would potentially create information security vulnerabilities that would not have been easy to identify at system design time.

These types of vulnerabilities pose challenges to the preservation of information security in organisations. Hence while most approaches to security requirements elicitation focus more on *technical* aspects of security, we have approached the problem from a *socio-technical* perspective in which we consider the behaviour of the participants and the effect on the system in terms of the security requirements of the system. We have presented our approach as a tool which can aid system designers and security engineers for socio-technical systems in the elicitation of security requirements for the system-to-be.

However, the work presented in this thesis is not without its limitations. We point out some of these limitations as follows;

1. Security goals which guide the elicitation of security requirements have been traditionally categorised into Confidentiality, Integrity and Availability. We have only applied our approach to Confidentiality and Integrity scenarios. At this stage we are not able to model Availability scenarios since the properties for availability can better be analysed dynamically than statically. We also want to point out that we have treated these security properties in isolation, that is, we have not given attention to how these goals affect each other.

2. We have not applied our approach to a real life organisational information security problem, even though we make an illustration using the iTrust medical record system. Therefore we expect that some issues may arise when applied to a real life scenario which we have not put into consideration.

3. So far, we have focused only on single institutions which means that we can only deal with security requirements that pertain to a single organisational setting. In order to analyse security requirements across organisational boundaries, it would require multiple institutions which will also have to handle the issues that would arise such as conflicts in requirements. Some published work on coordinating conflicts in interacting institutions (Lupu and Sloman, 1999; Li, 2013; Li et al., 2013) suggest that this is feasible.

4. There is a lack of a form of integration in which the various tools presented in this thesis may be used seamlessly. Each tool has to be executed separately. The provision of an integration platform would enhance the use of the tools.

We hope to be able to address these limitations in future works.

Bibliography

The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings, 2002. ACM.

Fighting to Close the Gap: Ernst & Young's 2012 Global Information Security Survey, November 2012. URL `http://www.ey.com/Publication/vwLUAssets/Fighting_to_close_the_gap:_2012_Global_Information_Security_Survey/$FILE/2012_Global_Information_Security_Survey___Fighting_to_close_the_gap.pdf`. Accessed 10.02.2013.

Camille Ben Achour, Colette Rolland, NAM Maiden, and C Souveyet. Guiding use case authoring: results of an empirical study. In *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, pages 36–43. IEEE, 1999.

Huib Aldewereld. Autonomy vs. conformity: An institutional perspective on norms and protocols. *The Knowledge Engineering Review*, 24(04):410–411, 2009.

Ian Alexander. Misuse cases: use cases with hostile intent. *Software, IEEE*, 20(1): 58–66, 2003. ISSN 0740-7459.

Ian Alexander and Neil Maiden, editors. *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. John Wiley & Sons Ltd, 2004.

ALIVE. ALIVE Project FP7-215890. ALIVE Deliverable D2.1: State of the Art. Technical report, 2010. URL `http://www.ist-alive.eu`. Accessed April 2010.

James F. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6(4):341–355, 1991. ISSN 1098-111X. doi: 10.1002/int.4550060403. URL `http://dx.doi.org/10.1002/int.4550060403`.

Robert H. Anderson and Richard Brackney. Understanding the insider threat: Pro-

ceedings of a March 2004 workshop. Technical report, RAND Corporation, Santa Monica, CA, March 2004.

Ross Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS '93, pages 215–227, New York, NY, USA, 1993. ACM. ISBN 0-89791-629-8. doi: 10.1145/168588.168615. URL http://doi.acm.org/10.1145/168588.168615.

Ross J. Anderson. *Security Engineering*. Wiley Publishing, Inc., second edition edition, 2008.

G. Aldo Antonelli. Non-monotonic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2012 edition, 2012.

Masahiko Aoki. *Towards a Comparative Institutional Analysis*. MIT Press, 2001.

Jim Arlow. Use cases, UML visual modelling and the trivialisation of business requirements. *Requirements Engineering*, 3(2):150–152, 1998.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi: DOI:10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4):425–461, 2003.

Tina Balke. *Towards the Governance of Open Distributed Systems: A Case Study in Wireless Mobile Grids*. CreateSpace, 2011.

A.K. Bandara, E.C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 26 – 39, June 2003. doi: 10.1109/POLICY.2003.1206955.

Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.

Chitta Baral. Reasoning about Actions and Change: From Single Agent Actions to Multi-agent Actions. In *Proceedings of the Twelfth International Conference on the*

Principles of Knowledge Representation and Reasoning (KR 2010). Association for the Advancement of Artificial Intelligence, 2010.

Chitta Baral, Gregory Gelfond, Tran Cao Son, and Enrico Pontelli. Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 259–266. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

Adam Beautement, M. Angela Sasse, and Mike Wonham. The compliance budget: managing security behaviour in organisations. In *Proceedings of the 2008 workshop on New security paradigms*, NSPW '08, pages 47–58, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-341-9. doi: 10.1145/1595676.1595684. URL `http://doi.acm.org/10.1145/1595676.1595684`.

Christoph Beierle, Oliver Dusso, and Gabriele Kern-Isberner. Using answer set programming for a decision support system. In *Logic Programming and Nonmonotonic Reasoning*, pages 374–378. Springer, 2005.

Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 2010.

Yolanta Beres, Marco Casassa Mont, Jonathan Griffin, and Simon Shiu. Using security metrics coupled with predictive modeling and simulation to assess security processes. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 564–573. IEEE Computer Society, 2009.

Gideon Bibu and Julian Padget. Security Engineering for Multi-Agent Systems: A Normative Approach (Poster). In *The 10th International Conference on Autonomous Agents and Multiagent Systems*, Taipei International Convention Center (TICC) in Taipei, Taiwan, May 2011. International Foundation for Autonomous Agents and Multiagent Systems.

Gideon Bibu, Nobukazu Yoshioka, and Julian Padget. System security requirements analysis with answer set programming. In *Requirements Engineering for Systems, Services and Systems-of-Systems (RES4), 2012 IEEE Second Workshop on*, pages 10–13, 2012a. doi: 10.1109/RES4.2012.6347689.

Gideon D. Bibu. Security in the context of multi-agent systems. In *The 10th In-*

ternational Conference on Autonomous Agents and Multiagent Systems - Volume 3, AAMAS '11, pages 1339–1340, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9826571-7-X, 978-0-9826571-7-1. URL http://dl.acm.org/citation.cfm?id=2034396.2034555.

Gideon D. Bibu, Nobukazu Yoshioka, and Julian A. Padget. Security Requirements Analysis and Validation using Misuse Case and Institutional Framework. Technical Report GRACE-TR 2012-02, GRACE Center of National Institute of Informatics, 2012b. URL http://grace-center.jp/rsc_tr-html?lang=en. Accessed 10.01.2013.

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

Matt Bishop. Position: "Insider" is Relative. In *Proceedings of the 2005 Workshop on New Security Paradigms*, NSPW '05, pages 77–78, New York, NY, USA, 2005. ACM. ISBN 1-59593-317-4. doi: 10.1145/1146269.1146288. URL http://doi.acm.org/10.1145/1146269.1146288.

Matt Bishop and Carrie Gates. Defining the insider threat. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, page 15. ACM, 2008.

Olivier Bodenreider, Zeynep H Coban, Mahir C Doganay, Esra Erdem, and Hilal Kosucu. A preliminary report on answering complex queries related to drug discovery using answer set programming. *ALPSWS 2008: Applications of Logic Programming to the (Semantic) Web and Web Services*, page 85, 2008.

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons Ltd, 2007.

Thouraya Bouabana-Tebibel and Mounira Belmesk. An Object-oriented Approach to Formally Analyze the UML 2.0 Activity Partitions. *Information and Software Technology*, 49(9-10):999–1016, September 2007. ISSN 0950-5849. doi: 10.1016/j.infsof.2006.10.007. URL http://dx.doi.org/10.1016/j.infsof.2006.10.007.

Gary Bouma. Distinguishing institutions and organisations in social change. *Journal of Sociology*, 34(3):232–245, 1998.

Fabricio A. Braz, Eduardo B. Fernandez, and Michael VanHilst. Eliciting Security Requirements through Misuse Activities. *Database and Expert Systems Applications, International Workshop on*, 0:328–333, 2008. ISSN 1529-4188. doi: http://doi.ieeecomputersociety.org/10.1109/DEXA.2008.101.

Frances Brazier, Virginia Dignum, Michael N. Huhns, Christian Derksen, Frank Dignum, Tim Lessner, Julian Padget, Thomas Quillinan, and Munindar P. Singh. Agent-based organisational governance of services. *Multiagent and Grid Systems*, 8 (1):3–18, Jan 2012. doi: 10.3233/MGS-2012-0187. URL `http://dx.doi.org/10.3233/MGS-2012-0187`.

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54:92–103, December 2011. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/2043174.2043195. URL `http://doi.acm.org/10.1145/2043174.2043195`.

Burcu Bulgurcu, Hasan Cavusoglu, and Izak Benbasat. Analysis of Perceived Burden of Compliance: The Role of Fairness, Awareness, and Facilitating Conditions. In *Association of Information Systems SIGSEC Workshop on Information Security & Privacy (WISP 2008). December 13, 2008, Paris, France*, 2008.

Burcu Bulgurcu, Hasan Cavusoglu, and Izak Benbasat. Information security policy compliance: an empirical study of rationality-based beliefs and information security awareness. *MIS quarterly*, 34(3):523–548, 2010.

Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, and David L Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 46–51. IEEE, 1990.

Michael Burrows, Martin Abadi, and Roger Needham. A Logic of Authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, February 1990. ISSN 0734-2071. doi: 10.1145/77648.77649. URL `http://doi.acm.org/10.1145/77648.77649`.

M. Camilli. Petri nets state space analysis in the cloud. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1638–1640, June 2012. doi: 10.1109/ICSE.2012.6227217.

John M Carroll. *Scenario-based design: envisioning work and technology in system development*. John Wiley and Sons, 1995.

Robert B. Cialdini. Crafting Normative Messages to Protect the Environment. *Current Directions in Psychological Science*, 12(4):105–109, 2003. doi: 10.1111/1467-8721.01242. URL `http://cdp.sagepub.com/content/12/4/105.abstract`.

Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

William J. Clancey, Patricia Sachs, Maarten Sierhuis, and Ron van Hoof. Brahms: simulating practice for work systems design. *Int. J. Hum.-Comput. Stud.*, 49(6): 831–865, 1998.

Owen Cliffe. *Specifying and analysing institutions in multi-agent systems using answer set programming*. Department of Computer Science, University of Bath, 2007.

Owen Cliffe, Marina De Vos, and Julian Padget. *Specifying and analysing agent-based social institutions using answer set programming*. Springer, 2006.

Owen Cliffe, Marina De Vos, and Julian Padget. Answer Set Programming for Representing and Reasoning About Virtual Institutions. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *Computational Logic in Multi-Agent Systems*, volume 4371 of *Lecture Notes in Computer Science*, pages 60–79. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-69619-3_4. URL `http://dx.doi.org/10.1007/978-3-540-69619-3_4`.

Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

Carl Colwill. Human factors in information security: The insider threat: Who can you trust these days? *Information Security Technical Report*, 14(4):186 – 196, 2009. ISSN 1363-4127. doi: http://dx.doi.org/10.1016/j.istr.2010.04.004. URL `http://www.sciencedirect.com/science/article/pii/S1363412710000051`. Human Factors in Information Security.

Robert Cooper and Michael Foster. Sociotechnical systems. *American Psychologist*, 26 (5):467, 1971.

Luciano R. Coutinho, Jaime S. Sichman, and Olivier Boissier. Modeling Organization in MAS: A Comparison of Models. In *Proceedings of the 1st. Workshop on Software Engineering for Agent-Oriented Systems (SEAS05)*, 2005.

John Critchley. Encourage compliance through convenience, awareness and culture,

September 2009. URL http://johncritchley.com/enterprise-architecture/encourage-compliance-through-convenience-awareness-and-culture. Accessed 03.05.2013.

The Common Criteria. Common Criteria for Information Technology Security Evaluation, September 2012. URL http://www.commoncriteriaportal.org/cc/. Accessed 20.02.2013.

Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Adaptive socio-technical systems: a requirements-based approach. *Requirements Engineering*, 18(1):1–24, 2013. ISSN 0947-3602. doi: 10.1007/s00766-011-0132-1. URL http://dx.doi.org/10.1007/s00766-011-0132-1.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

Marina De Vos, Tom Crick, Julian Padget, Martin Brain, Owen Cliffe, and Jonathan Needham. LAIMA: A multi-agent platform using ordered choice logic programming. In *Declarative Agent Languages and Technologies III*, pages 72–88. Springer, 2006.

Edward L. Deci and Richard M. Ryan. The support of autonomy and the control of behav. *Journal of personality and social psychology*, 53(6):1024–1037, 1987. doi: 10.1037/0022-3514.53.6.1024. URL http://psycnet.apa.org/journals/psp/53/6/1024.pdf.

JamesP. Delgrande, Torsten Grote, and Aaron Hunter. A General Approach to the Verification of Cryptographic Protocols Using Answer Set Programming. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 355–367. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_30. URL http://dx.doi.org/10.1007/978-3-642-04238-6_30.

Scott A. DeLoach, Mark F. Wood, and Clint H. Sparkman. Multiagent Systems Engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of ECAI*, volume 92, pages 384–388. Citeseer, 1992.

Frank Dignum, Hans Weigand, and Egon Verharen. Meeting the deadline: on the formal specification of temporal deontic constraints. In *Foundations of Intelligent Systems*, pages 243–252. Springer, 1996.

Virginia Dignum. *A Model for Organisational Interaction: Based on Agents, Founded in Logic*. PhD thesis, Utretch University, Netherlands, 2004.

Virginia Dignum, Javier Vázquez-Salceda, and Frank Dignum. OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *PROMAS*, volume 3346 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2004. ISBN 3-540-24559-6.

DofBIS-UK. 2013 Information Security Breaches Survey. Technical report, Department for Business Innovation & Skills, UK, 2013. URL `http://www.pwc.co.uk/assets/pdf/cyber-security-2013-technical-report.pdf`. http://www.pwc.co.uk/assets/pdf/cyber-security-2013-technical-report.pdf Accessed July 1, 2013.

Marlon Dumas and Arthur H. M. ter Hofstede. UML Activity Diagrams As a Workflow Specification Language. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 76–90, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42667-1. URL `http://dl.acm.org/citation.cfm?id=647245.719456`.

Wolfgang Dzida and Regine Freitag. Making use of scenarios for validating analysis and design. *Software Engineering, IEEE Transactions on*, 24(12):1182–1196, 1998.

Deborah East and Miroslaw Truszczyński. Predicate-calculus-based Logics for Modeling and Solving Search Problems. *ACM Trans. Comput. Logic*, 7(1):38–83, January 2006. ISSN 1529-3785. doi: 10.1145/1119439.1119441. URL `http://doi.acm.org/10.1145/1119439.1119441`.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1):99–111, 1999.

Thomas Eiter, Gerhard Brewka, Minh Dao-Tran, Michael Fink, Giovambattista Ianni,

and Thomas Krennwallner. Combining nonmonotonic knowledge bases with external sources. In *Frontiers of Combining Systems*, pages 18–42. Springer Berlin Heidelberg, 2009.

Mohamed El-Attar. From misuse cases to mal-activity diagrams: bridging the gap between functional security analysis and design. *Software & Systems Modeling*, pages 1–18, 2012a. ISSN 1619-1366. doi: 10.1007/s10270-012-0240-5. URL `http://dx.doi.org/10.1007/s10270-012-0240-5`.

Mohamed El-Attar. Towards developing consistent misuse case models. *Journal of Systems and Software*, 85(2):323–339, 2012b.

Esra Erdem and Reyyan Yeniterzi. Transforming controlled natural language biomedical queries into answer set programs. In *Proceedings of the Workshop on Current Trends in Biomedical Natural Language Processing*, pages 117–124. Association for Computational Linguistics, 2009.

Esra Erdem, Yelda Erdem, Halit Erdogan, and Umut Öztok. Finding Answers and Generating Explanations for Complex Biomedical Queries. In *Proc. of AAAI*, 2011.

Rik Eshuis and Roel Wieringa. A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models. Technical Report TR-CTIT-01-04, Enschede, 2001. URL `http://doc.utwente.nl/37504/`.

Marc Esteva, Juan A. Rodríguez-Aguilar, Josep Lluís Arcos, Carles Sierra, and Pere Garcia. Formalising Agent Mediated Electronic Institutions. In *Agent Mediated Electronic Commerce, LNAI 1991*, pages 29–38. Springer-Verlag, 2000.

Marc Esteva, Juan-Antonio Rodriguez-Aguilar, Carles Sierra, Pere Garcia, and Josep L Arcos. On the formal specification of electronic institutions. In *Agent mediated electronic commerce*, pages 126–147. Springer, 2001.

Marc Esteva, David de la Cruz, and Carles Sierra. ISLANDER: an electronic institutions editor. In *AAMAS* DBL (2002), pages 1045–1052.

Marc Esteva, Bruno Rosell, Juan A. Rodríguez-Aguilar, and Josep Lluís Arcos. AMELI: An Agent-Based Middleware for Electronic Institutions. In *AAMAS*, pages 236–243. IEEE Computer Society, 2004. ISBN 1-58113-864-4.

Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In Wolf Zimmer-

mann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22094-7. doi: 10.1007/978-3-540-24773-9_7. URL http://dx.doi.org/10.1007/978-3-540-24773-9_7.

Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From Agents to Organizations: An Organizational View of Multi-agent Systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *AOSE*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer, 2003. ISBN 3-540-20826-7.

Eduardo Fernandez, Michael VanHilst, Maria Larrondo Petrie, and Shihong Huang. Defining Security Requirements Through Misuse Actions. In Sergio Ochoa and Gruia-Catalin Roman, editors, *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, volume 219 of *IFIP International Federation for Information Processing*, pages 123–137. Springer Boston, 2006. ISBN 978-0-387-34828-5. doi: 10.1007/978-0-387-34831-5_10. URL http://dx.doi.org/10.1007/978-0-387-34831-5_10.

Donald Firesmith. Use Case Modeling Guidelines. In Donald Firesmith, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *TOOLS (30)*, pages 184–193. IEEE Computer Society, 1999. ISBN 0-7695-0278-4.

Donald Firesmith. Security Use Cases. *Journal of Object Technology*, 2(1):53–64, 2003.

N. Fornara, H.L. Cardoso, P. Noriega, E. Oliveira, C. Tampitsikas, and M.I. Schumacher. Modelling Agent Institutions. *Agreement Technologies*, page 277, 2013.

Stroz Friedberg. On the pulse: Information security risk in American business. Survey report, Stroz Friedberg, November 2013. URL http://www.strozfriedberg.com/wp-content/uploads/2014/01/Stroz-Friedberg_On-the-Pulse_Information-Security-in-American-Business.pdf. Accessed 25 February 2014.

Chris Fry and Martin Nystrom. *Security Monitoring*. O'Reilly Media, Inc., 1st edition, February 2009.

Juan C. García-Ojeda, Scott A. DeLoach, Robby, Walamitien H. Oyenan, and Jorge Valenzuela. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. In Michael Luck and Lin Padgham, editors, *AOSE*, volume 4951

of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007. ISBN 978-3-540-79487-5.

Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold, New York, USA, 1988.

Martin Gebser, R Kaminiski, Benjamin Kaufmann, M Ostrowsky, Torsten Schaub, and Sven Thiele. Using gringo, clingo and iclingo, 2008. URL `http://cs.swan.ac.uk/~csoliver/ok-sat-library/OKplatform/ExternalSources/sources/SAT/Potassco/GringoManual.pdf`. Accessed 30.10.2011.

Martin Gebser, Roland Kaufmann, and Torsten Schaub. Correct Reasoning. chapter Gearing Up for Effective ASP Planning, pages 296–310. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 978-3-642-30742-3. URL `http://dl.acm.org/citation.cfm?id=2363344.2363364`.

Michael Gelfond. Answer Sets. In Frank van Harmelen, Vladimír Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 285–316. Elsevier, 2008.

Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics For Logic Programming. pages 1070–1080. MIT Press, 1988.

Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2):301–321, 1993.

Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of the National Conference on Artificial Intelligence*, pages 623–630. Citeseer, 1998.

Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Causal laws and multi-valued fluents. In *Proceedings of Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)*. Citeseer, 2001.

Janice Glasgow, Glenn Macewen, and Prakash Panangaden. A logic for reasoning about security. *ACM Trans. Comput. Syst.*, 10(3):226–264, August 1992. ISSN 0734-2071. doi: 10.1145/146937.146940. URL `http://doi.acm.org/10.1145/146937.146940`.

Martin Glinz. On Non-Functional Requirements. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 21 –26, oct. 2007. doi: 10.1109/RE.2007.45.

Alvin I. Goldman. *A Theory of Human Action*. Princeton Univ Press, 1976.

Chloe Green. Employees revealed as greatest challenge to IT security. *Information Age*, January 2014. URL `http://www.information-age.com/technology/security/123457586/employees-revealed-as-greatest-challenge-to-it-security`. Accessed 28 February 2014.

Frank L Greitzer, Andrew P Moore, Dawn M Cappelli, Dee H Andrews, Lynn A Carroll, and Thomas D Hull. Combating the insider cyber threat. *Security & Privacy, IEEE*, 6(1):61–64, 2008.

N. Guelfi and A. Mammar. A formal semantics of timed activity diagrams and its PROMELA translation. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 8 pp.–, Dec 2005. doi: 10.1109/APSEC.2005.7.

Volker Haarslev and Ralf Müller. RACER system description. In *Automated Reasoning*, pages 701–705. Springer, 2001.

Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar Nuseibeh. A Framework for Security Requirements Engineering. In *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, SESS '06, pages 35–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-411-1. doi: 10.1145/1137627.1137634. URL `http://doi.acm.org/10.1145/1137627.1137634`.

Mahdi Hannoun, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. MOISE: An Organizational Model for Multi-agent Systems. In Maria Carolina Monard and Jaime Simão Sichman, editors, *IBERAMIA-SBIA*, volume 1952 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2000. ISBN 3-540-41276-X.

John Hawes. Who's to blame for security problems? Surveys say, EVERYONE. *nakedsecurity*, January 2014. URL `http://nakedsecurity.sophos.com/2014/01/15/whos-to-blame-for-security-problems-surveys-say-everyone/`. Accessed 28 February 2014.

Morten Hertzum. Making use of scenarios: a field study of conceptual design. *International Journal of Human-Computer Studies*, 58(2):215–239, 2003.

Risto Hilpinen. *Deontic logic: Introductory and systematic readings*, volume 33. D Reidel Publishing Company, 1971.

Gerard J Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

Luke Hopton, Owen Cliffe, Marina De Vos, and Julian Padget. InstQL: a query language for virtual institutions using answer set programming. In *Proceedings of the 10th international conference on Computational logic in multi-agent systems*, CLIMA'09, pages 102–121, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16866-3, 978-3-642-16866-6. URL `http://dl.acm.org/citation.cfm?id=1927368.1927374`.

Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. MOISE+: towards a structural, functional, and deontic model for MAS organization. In *AAMAS* DBL (2002), pages 501–502.

Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. S-MOISE+: A Middleware for Developing Organised Multi-agent Systems. In Olivier Boissier, Julian A. Padget, Virginia Dignum, Gabriela Lindemann, Eric T. Matson, Sascha Ossowski, Jaime Simão Sichman, and Javier Vázquez-Salceda, editors, *AAMAS Workshops*, volume 3913 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005. ISBN 3-540-35173-6.

Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Developing Organised Multi-Agent Systems Using the Moise+ Model: Programming Issues at the System and Agent Levels. *IJAOSE*, 1(3/4):370–395, 2007.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990. URL `http://dx.doi.org/10.1109/IEEESTD.1990.101064`. Accessed 20.2.2011.

Ponemon Institute. The human factor in data protection. Technical report, Ponemon Institute, January 2012. URL `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt_trend-micro_ponemon-survey-2012.pdf`. Accessed 10/01/2014.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Pearson Education, 1999.

M. Eric Johnson and Eric Goetz. Embedding Information Security into the Organization. *IEEE Security and Privacy*, 5(3):16–24, May 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.59. URL `http://dx.doi.org/10.1109/MSP.2007.59`.

Andrew J. I. Jones and Marek Sergot. Formal specification of security requirements using the theory of normative positions. In Yves Deswarte, Grard Eizenberg, and Jean-Jacques Quisquater, editors, *Computer Security ESORICS 92*, volume 648 of *Lecture Notes in Computer Science*, pages 103–121. Springer Berlin Heidelberg, 1992. ISBN 978-3-540-56246-7. doi: 10.1007/BFb0013894. URL `http://dx.doi.org/10.1007/BFb0013894`.

Andrew J.I. Jones and Marek Sergot. A Formal Characterisation of Institutionalised Power. *ACM Computing Surveys*, 28(4es):121, 1996. URL `http://www-lp.doc.ic.ac.uk/_lp/Sergot/InstitPower.ps.gz`.

Arie Karniel and Yoram Reich. Process Modeling Using Workflow-Nets. In *Managing the Dynamics of New Product Development Processes*, pages 75–95. Springer London, 2011. ISBN 978-0-85729-569-9. doi: 10.1007/978-0-85729-570-5_6. URL `http://dx.doi.org/10.1007/978-0-85729-570-5_6`.

Richard Kissel, Kevin Stine, Matthew Scholl, Hart Rossman, Jim Fahlsing, and Jessica Gulick. Security Considerations in the System Development Life Cycle. Technical Report 800-64 Revision 2, NIST, October 2008. URL `http://csrc.nist.gov/publications/nistpubs/800-64-Rev2/SP800-64-Revision2.pdf`. Accessed 7 February 2012.

Christoph Knieke and Ursula Goltz. An Executable Semantics for UML 2 Activity Diagrams. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, FML '10, pages 3:1–3:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0532-7. doi: 10.1145/1943397.1943400. URL `http://doi.acm.org/10.1145/1943397.1943400`.

Gerald Kotonya and Ian Sommerville. *Requirements Engineering: processes and techniques*. John Wiley and Sons, 1998.

Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

Sara Kraemer, Pascale Carayon, and John Clem. Human and organizational factors in computer and information security: Pathways to vulnerabilities. *Computers & Security*, 28(7):509 – 520, 2009. ISSN 0167-4048. doi: http://dx.doi.org/10.1016/j.cose.2009.04.006. URL `http://www.sciencedirect.com/science/article/pii/S0167404809000467`.

Lars M. Kristensen. A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future. In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 39–42. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13674-0. doi: 10.1007/978-3-642-13675-7_4. URL `http://dx.doi.org/10.1007/978-3-642-13675-7_4`.

Per Kroll and Philippe Kruchten. *The rational unified process made easy: a practitioner's guide to the RUP*. Addison-Wesley Professional, 2003.

Daryl Kulak and Eamonn Guiney. *Use cases: requirements in context*. Addison-Wesley Professional, 2004.

Charles Lakos and Laure Petrucci. Modular state space exploration for timed Petri nets. *International Journal on Software Tools for Technology Transfer*, 9(3-4):393–411, 2007.

Vitus SW Lam. A Formalism for Reasoning about UML Activity Diagrams. *Nord. J. Comput.*, 14(1-2):43–64, 2007.

Younghwa Lee, Jintae Lee, and Zoonky Lee. Integrating Software Lifecycle Process Standards with Security Engineering. *Computers & Security*, 21(4):345 – 355, 2002. ISSN 0167-4048. doi: 10.1016/S0167-4048(02)00413-3. URL `http://www.sciencedirect.com/science/article/pii/S0167404802004133`.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Christoph Koch, et al. The DLV System. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 537–540. Springer-Verlag, 2002.

Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998.

Tingting Li. Normative Conflict Detection and Resolution in Cooperating Institutions. In Francesca Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013. ISBN 978-1-57735-633-2.

Tingting Li, Tina Balke, Marina De Vos, Julian Padget, and Ken Satoh. Legal Conflict Detection in Interacting Legal Systems. In Kevin D. Ashley, editor, *JURIX*, volume 259 of *Frontiers in Artificial Intelligence and Applications*, pages 107–116. IOS Press, 2013. ISBN 978-1-61499-358-2, 978-1-61499-359-9.

Yuliya Lierler. cmodels–SAT-based disjunctive answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, pages 447–451. Springer, 2005.

Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Logic Programming and Nonmonotonic Reasoning*, pages 346–350. Springer, 2004.

Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002.

Vladimir Lifschitz. What Is Answer Set Programming?. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, volume 8, pages 1594–1597, 2008. URL `http://www.aaai.org/Papers/AAAI/2008/AAAI08-270.pdf`.

Susan Lilly. Use case pitfalls: top 10 problems from real projects using use cases. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 174–183. IEEE, 1999.

Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

L. Liu, E. Yu, and J. Mylopoulos. Security and privacy requirements analysis within a social setting. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 151 – 161, sept. 2003. doi: 10.1109/ICRE.2003.1232746.

Gert-Jan Lokhorst. Mallyś Deontic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2008 edition, 2008.

Alessio Lomuscio and Franco Raimondi. MCMAS: A model checker for multi-agent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–454. Springer, 2006.

Emil C Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852–869, 1999.

Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation*, pages 2–15. Winter Simulation Conference, 2005.

Ernst Mally. *Grundgesetze des Sollens: Elemente der Logik des Willens*. Graz: Leuschner und Lubensky, Universitäts-Buchhandlung, 1926.

Wiktor Marek and Miroslaw Truszczyński. Autoepistemic logic. *J. ACM*, 38(3):587–618, July 1991. ISSN 0004-5411. doi: 10.1145/116825.116836. URL `http://doi.acm.org/10.1145/116825.116836`.

Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of the National Conference on Artificial Intelligence*, pages 460–465. John Wiley & Sons LTD, 1997.

John McCarthy. Situations, actions, and causal laws. Technical report, DTIC Document, 1963. URL `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0785031`. Accessed 12.06.2013.

John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980.

John McCarthy and Patrick Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

J. McDermott and C. Fox. Using abuse case models for security requirements analysis. In *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual*, pages 55 –64, 1999. doi: 10.1109/CSAC.1999.816013.

Peter Mell and Timothy Grance. The NIST definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011. URL `http://www.smaele.nl/edocs/NIST-Definition-Of-Cloud-Computing-2011.pdf`. Accessed January 10 2014.

Stephan Merz. An introduction to model checking. *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, pages 77–110, 2008.

Mike and Kemp. Barbarians inside the gates: addressing internal security threats. *Network Security*, 2005(6):11 – 13, 2005. ISSN 1353-4858. doi:

10.1016/S1353-4858(05)70247-6. URL http://www.sciencedirect.com/science/article/pii/S1353485805702476.

Rob Miller and Murray Shanahan. Some Alternative Formulations of the Event Calculus. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 452–490, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43960-9. URL http://dl.acm.org/citation.cfm?id=646002.675932.

Naftaly H. Minsky and Abe D. Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 92–102, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. ISBN 0-8186-0620-7. URL http://dl.acm.org/citation.cfm?id=319568.319589.

Mehdi Mirakhorli and Jane Cleland-Huang. Tracing Non-Functional Requirements. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 299–320. Springer London, 2012. ISBN 978-1-4471-2238-8. doi: 10.1007/978-1-4471-2239-5_14. URL http://dx.doi.org/10.1007/978-1-4471-2239-5_14.

Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. Wiley Publishing, Inc., 2002.

Jonathan D. Moffett and Bashar A. Nuseibeh. A Framework for Security Requirements Engineering. 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.607. Accessed 12.12.2012.

Haralambos Mouratidis. Secure Software Systems Engineering: The Secure Tropos Approach (Invited Paper). *JSW*, 6(3):331–339, 2011.

Haralambos Mouratidis, Paolo Giorgini, and Gordon Manson. Integrating Security and Systems Engineering: Towards the Modelling of Secure Information Systems. In Johann Eder and Michele Missikoff, editors, *Advanced Information Systems Engineering*, volume 2681 of *Lecture Notes in Computer Science*, pages 63–78. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40442-2. doi: 10.1007/3-540-45017-3_7. URL http://dx.doi.org/10.1007/3-540-45017-3_7.

Haralambos Mouratidis, Paolo Giorgini, and Gordon Manson. Towards the development of secure information systems: Security Reference Diagram and Security Attack

Scenarios. In *Proceedings of the FORUM at International Conference on Advanced Information Systems, Riga Latvia*, 2004.

Erik T Mueller. Event calculus reasoning through satisfiability. *Journal of Logic and Computation*, 14(5):703–730, 2004.

Erik T Mueller. Event Calculus. *Handbook of Knowledge Representation*, page 671, 2008.

Ilkka Niemelä and Patrik Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *LPNMR*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997. ISBN 3-540-63255-7.

Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *Practical Aspects of Declarative Languages*, pages 169–183. Springer Berlin Heidelberg, 2001.

Douglass C. North. *Institutions, Institutional Change and Economic Performance*. Cambridge University Press, 1990.

Daniel Okouya and Virginia Dignum. OperettA: a prototype tool for the design, analysis and development of multi-agent organizations. In *AAMAS (Demos)*, pages 1677–1678. IFAAMAS, 2008.

T. Okubo, K. Taguchi, and N. Yoshioka. Misuse Cases + Assets + Security Goals. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 3, pages 424 –429, aug. 2009. doi: 10.1109/CSE.2009.18.

T. Okubo, H. Kaiya, and N. Yoshioka. Effective Security Impact Analysis with Patterns for Software Enhancement. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 527 –534, aug. 2011. doi: 10.1109/ARES.2011.79.

Martin S. Olivier. Database Privacy: Balancing Confidentiality, Integrity and Availability. *ACM SIGKDD Explorations*, 4(2):20–27, 2002.

OMG. OMG Unified Modelling Language (OMG UML), Superstructure, August 2011. URL `http://www.omg.org/spec/UML/2.4.1/Superstructure`. Accessed December 27, 2013.

Elinor Ostrom. An agenda for the study of institutions. *Public Choice*, 48:3–25, 1986. ISSN 0048-5829. doi: 10.1007/BF00239556. URL http://dx.doi.org/10.1007/BF00239556.

Elda Paja, Fabiano Dalpiaz, Mauro Poggianella, Pierluigi Roberti, and Paolo Giorgini. Specifying and Reasoning over Socio-Technical Security Requirements with STS-Tool. In *32nd International Conference on Conceptual Modeling - Workshops (ER'13 Workshops) – to appear*, 2013. Available on http://www.sts-tool.eu/doc/publications/paja-dalp-pogg-robe-gior-13-er.pdf.

Mike Pastore and Emmett Dulaney. *CompTIA Security+ Study Guide*. Wiley Publishing, Inc., 3rd edition, 2006.

Joshua J. Pauli and Dianxiang Xu. Misuse Case-Based Design and Analysis of Secure Software Architecture. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 398–403, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2315-3. doi: 10.1109/ITCC.2005.199. URL http://dx.doi.org/10.1109/ITCC.2005.199.

Cyrus Peikari and Seth Fogie. Writing a security policy, July 2003. Available at http://searchsecurity.techtarget.com/tip/Writing-a-security-policy. Accessed April 25, 2014.

Andrew M. Pettigrew. On Studying Organizational Cultures. *Administrative Science Quarterly*, 24(4):pp. 570–581, 1979. ISSN 00018392. URL http://www.jstor.org/stable/2392363.

Wolter Pieters. The (Social) Construction of Information Security. *The Information Society*, 27(5):326–335, 2011. doi: 10.1080/01972243.2011.607038. URL http://www.tandfonline.com/doi/abs/10.1080/01972243.2011.607038.

Wolter Pieters and Lizzie Coles-Kemp. Reducing normative conflicts in information security. In *Proceedings of the 2011 workshop on New security paradigms workshop*, NSPW '11, pages 11–24, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1078-9. doi: 10.1145/2073276.2073279. URL http://doi.acm.org/10.1145/2073276.2073279.

Wolter Pieters, Trajce Dimkov, and Dusko Pavlovic. Security Policy Alignment: A Formal Approach. *IEEE Systems Journal*, 7:275–287, 2013.

Enrico Pontelli, Tran Cao Son, Chitta Baral, and Gregory Gelfond. Answer set programming and planning with knowledge and world-altering actions in multiple agent domains. In *Correct Reasoning*, pages 509–526. Springer, 2012.

Kevin Poulsen. Mitnick to Lawmakers: People, Phones are Weakest Links, March 2000. URL `http://www.politechbot.com/p-00969.html`.

Walter W. Powell and Paul J. DiMaggio, editors. *The New institutionalism in Organisational Analysis*. The University of Chicago Press, 1991.

Richard Power. CSI/FBI 2002 Computer Crime and Security Survey. *Computer Security Journal*, 18(2):7–30, 2002.

Kieran Poynter. Review of information security at HM Revenue and Customs, June 2008. URL `http:/www.hm-treasury.gov.uk/media/0/1/poynter_review250608.pdf`.

Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer, 2007.

Christian W Probst, René Rydhof Hansen, and Flemming Nielson. Where can an insider attack? In *Formal Aspects in Security and Trust*, pages 127–142. Springer, 2007.

Petri Puhakainen and Mikko Siponen. Improving employees' compliance through information systems security training: an action research study. *MIS Q.*, 34(4):757–778, December 2010. ISSN 0276-7783. URL `http://dl.acm.org/citation.cfm?id=2017496.2017502`.

PWC. Information Security Breaches Survey. Technical report, PricewaterhouseCoopers LLP, 2012. URL `http://docs.media.bitpipe.com/io_10x/io_102267/item_485940/PwC_ISBS%20technical%20report_2012.pdf`. Accessed 02 April 2013.

Thomas B. Quillinan, Frances Brazier, Huib Aldewereld, Frank Dignum, Virginia Dignum, Loris Penserini, and Niek Wijngaards. Developing Agent-based Organizational Models for Crisis Management. In Decker, Sichman, Sierra, and Castelfranchi, editors, *AAMAS 2009*. Budapest, Hungary, 2009.

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, Cambridge, Massachusetts, 2001.

Suzanne Robertson and James Robertson. *Mastering the Requirements Process (2nd Edition)*. Addison-Wesley Professional, 2006. ISBN 0321419499.

James Rumbaugh. Getting started-Using use cases to capture requirements. *Journal of Object-Oriented Programming*, 7(5):8, 1994.

John Rushby. Security Requirements Specifications: How and What? In *Symposium on Requirements Engineering for Information Security (SREIS)*, Indianapolis, IN, mar 2001. URL `http://www.csl.sri.com/papers/sreis01/`.

Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling - Volume 53*, APCCM '06, pages 95–104, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. ISBN 1-920-68235-X. URL `http://dl.acm.org/citation.cfm?id=1151855.1151866`.

Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

Chiaki Sakama. Dishonest reasoning by abduction. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, pages 1063–1068. AAAI Press, 2011.

August-wilhelm Scheer, Oliver Thomas, and Otmar Adam. Process modeling using event-driven process chains. In Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede, editors, *Process-Aware Information Systems: Bridging people and software through process technology*, pages 119–141. John Wiley & Sons Ltd, 2005.

Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley Publishing, Inc., 2000.

John R. Searle. *The Construction of Social Reality*. The Penguin Press, 1995.

M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The British Nationality Act as a logic program. *Commun. ACM*, 29(5):370–386, May 1986. ISSN 0001-0782. doi: 10.1145/5689.5920. URL `http://doi.acm.org/10.1145/5689.5920`.

Marek Sergot. $(\mathcal{C}+)^{++}$: An action language for modelling norms and institutions. Technical Report 2004/8, Department of Computing, Imperial College, London, 2005.

Murray Shanahan. Event calculus planning revisited. In *Recent Advances in AI Planning*, pages 390–402. Springer, 1997.

Murray Shanahan. Artificial intelligence today. chapter The event calculus explained, pages 409–430. Springer-Verlag, Berlin, Heidelberg, 1999. ISBN 3-540-66428-9. URL `http://dl.acm.org/citation.cfm?id=1805750.1805767`.

Carles Sierra, Juan A. Rodrguez-Aguilar, Pablo Noriega Blanco-Vigil, Josep-Llus Arcos-Rosell, and Marc Esteva-Vivancos. Engineering Multi-Agent Systems as Electronic Institutions. In *UPGRADE: The European Journal for the Informatics Professional*, volume Vol. V, No. 4. Novatica, 2004.

Guttorm Sindre. Mal-Activity Diagrams for Capturing Attacks on Business Processes. In Pete Sawyer, Barbara Paech, and Patrick Heymans, editors, *Requirements Engineering: Foundation for Software Quality*, volume 4542 of *Lecture Notes in Computer Science*, pages 355–366. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73030-9. URL `http://dx.doi.org/10.1007/978-3-540-73031-6_27`.

Guttorm Sindre and Andreas L. Opdahl. Eliciting Security Requirements by Misuse Cases. In *TOOLS (37)*, pages 120–131. IEEE Computer Society, 2000.

Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requir. Eng.*, 10(1):34–44, 2005.

Michelle Slatalla and Joshua Quittner. *Masters of deception: The gang that ruled cyberspace*. HarperCollins Publishers, 1995.

Timo Soininen and Ilkka Niemelä. Developing a Declarative Rule Language for Applications in Product Configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319. Springer-Verlag, 1999.

Tran Cao Son, Enrico Pontelli, and Chiaki Sakama. Logic programming for multiagent planning with negotiation. In *Logic Programming*, pages 99–114. Springer, 2009.

J Michael Spivey. *The Z notation: a reference manual. International Series in Computer Science*. Prentice-Hall, New York, NY, 1992.

Steve Stasiukonis. Social engineering, the USB way. *Dark Reading*, 7, 2006.

Harald Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *Visual Languages*

and Human Centric Computing, 2004 IEEE Symposium on, pages 235–242, Sept 2004. doi: 10.1109/VLHCC.2004.46.

Hannes Strass and Michael Thielscher. A language for default reasoning about actions. In *Correct Reasoning*, pages 527–542. Springer, 2012.

Kanags Surendran. Information Security: Nurturing a Security Conscious Workforce. http://www.knowledgeplatform.com/Content/Pdfs/Information-Security-White-Paper.pdf, April 2005.

Samuel TC Thompson. Helping the hacker? Library information, security, and social engineering. *Information Technology and Libraries*, 25(4):222–225, 2013.

Tim Thornburgh. Social Engineering: The "Dark Art". In *Proceedings of the 1st Annual Conference on Information Security Curriculum Development*, InfoSecCD '04, pages 133–135, New York, NY, USA, 2004. ACM. ISBN 1-59593-048-5. doi: 10.1145/1059524.1059554. URL `http://doi.acm.org/10.1145/1059524.1059554`.

Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design*, pages 1290–1299, 2003.

EL Trist and KW Bamforth. Some Social and Psychological Consequences of the Longwall Method. *Human relations*, 4:3–38, 1951.

Eric Trist. The evolution of socio-technical systems. *Occasional paper*, 2, 1981.

Phan Huy Tu, Tran Cao Son, Michael Gelfond, and A Ricardo Morales. Approximation of action theories and its application to conformant planning. *Artificial Intelligence*, 175(1):79–119, 2011.

W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998. doi: 10.1142/S0218126698000043. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218126698000043`.

Wil van der Aalst and Kees Max van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004.

Wil MP van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, pages 407–426. Springer Berlin Heidelberg, 1997.

W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639 – 650, 1999. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/S0950-5849(99)00016-6. URL `http://www.sciencedirect.com/science/article/pii/S0950584999000166`.

Kees van Hee, Natalia Sidorova, and Marc Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In *Applications and Theory of Petri Nets 2003*, pages 337–356. Springer Berlin Heidelberg, 2003.

A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 148 – 157, may 2004. doi: 10.1109/ICSE.2004.1317437.

Axel van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 5–19, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. doi: 10.1145/337180.337184. URL `http://doi.acm.org/10.1145/337180.337184`.

Javier Vázquez-Salceda. Thesis: The role of norms and electronic institutions in multi-agent systems applied to complex domains. The HARMONIA framework. *AI Commun.*, 16(3):209–212, 2003.

Javier Vázquez-Salceda and Frank Dignum. Modelling Electronic Organizations. In Vladimír Marík, Jörg P. Müller, and Michal Pechoucek, editors, *CEEMAS*, volume 2691 of *Lecture Notes in Computer Science*, pages 584–593. Springer, 2003. ISBN 3-540-40450-3.

Willem Visser and Howard Barringer. Practical CTL* model checking: Should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2 (4):350–365, 2000.

Valdis Vitolins and Audris Kalnins. Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 181–192, Sept 2005. doi: 10.1109/EDOC.2005.29.

Georg Henrik von Wright. Deontic Logic. *Mind*, 60:1–15, 1951.

Erich Vranes. The Definition of Norm Conflict in International Law and Legal Theory.

European Journal of International Law, 17(2):395–418, 2006. doi: 10.1093/ejil/chl002. URL http://ejil.oxfordjournals.org/content/17/2/395.abstract.

K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: current practice. *Software, IEEE*, 15(2):34–45, 1998. ISSN 0740-7459. doi: 10.1109/52.663783.

Michael E. Whitman. Enemy at the Gate: Threats to Information Security. *Commun. ACM*, 46(8):91–95, August 2003. ISSN 0001-0782. doi: 10.1145/859670.859675. URL http://doi.acm.org/10.1145/859670.859675.

Michael E. Whitman and Herbert J. Mattord. *Principles of Information Security*. Cengage Learning, Boston, MA, USA, 4th edition, 2012. ISBN 9781111138219. URL http://books.google.co.uk/books?id=L3LtJAxcsmMC.

Michael E. Whitman, Anthony M. Townsend, and Robert J. Aalberts. Information Systems Security and the Need for Policy. In *Information Security Management: Global Challenges in the New Millennium*, pages 9–18. IGI Global, 2001. URL http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-878289-78-0.ch002.

Karl Eugene Wiegers. *Software Requirements*. Microsoft Press, 2003.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Laurie Williams, Tao Xie, Andy Meneely, Lauren Hayward, and Jason King. iTrust Medical Care Requirements Specification, October 2011. URL http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirements. Accessed 10 November 2012.

Ira S Winkler and Brian Dealy. Information security technology?... Dont rely on it. A case study in social engineering. In *Proceedings of the 5th usenix Unix Security Symposium. The usenix Association*, 1995.

Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the suitability of BPMN for business process modelling. In *Business Process Management*, pages 161–176. Springer Berlin Heidelberg, 2006.

Michael J. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, 2000.

Dong Xu, Nduwimfura Philbert, Zongtian Liu, and Wei Liu. Towards Formalizing

UML Activity Diagrams in CSP. In *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, volume 2, pages 450–453, 2008. doi: 10.1109/ISCSCT.2008.379.

D Yi-zhi, Yan-zhang Wang, and Yun-fei Liu. The formal semantics of an UML activity diagram. *Journal of Shanghai University (English Edition)*, 8(3):322–327, 2004.

Eric Yu, Paolo Giorgini, Neil Maiden, and John Mylopoulos, editors. *Social Modeling For Requirements Engineering*. MIT Press, 2011.

Nicola Zannone. The SI* modeling framework: metamodel and applications. *International Journal of Software Engineering and Knowledge Engineering*, 19(05):727–746, 2009. doi: 10.1142/S0218194009004374. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218194009004374`.

Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997. ISSN 1049-331X. doi: 10.1145/237432.237434. URL `http://doi.acm.org/10.1145/237432.237434`.

Appendices

Appendix A

XML2InstAL Translator Code

Listing A.1: XML2InstAL translator Code.

```
1  from xml.etree import ElementTree as et
2  from collections import OrderedDict as od
3
4
5  # create empty tables
6  actorTable = od()
7  actionTable = od()
8  eventTable = od()
9
10 def insertActor(actorId, actorName):
11     if actorTable.has_key(actorId):
12         print 'error! Actor already in dictionary, refuse to enter
               twice.'
13     else:
14         actorTable[actorId] = {
15             'actorName':actorName,
16             'actionIDs':[]
17             }
18     #print actorTable[actorId]
19
20 def insertAction(actionId, name):
21     if actionTable.has_key(actionId):
22         print 'error!'
23     else:
24         actionTable[actionId] = {
25             'actionType': name
```

```
26            }
27 # print actionTable[actionId]
28
29 def insertEvent(ID, name):
30     if eventTable.has_key(ID):
31         print 'error!'
32     else:
33         eventTable[ID] = {
34             'actionID':ID,
35             'From':[],
36             'To': [],
37             'actionType':name
38             }
39
40 tree = et.parse('examsProject2.xml')
41
42 #populate actorTable
43 for node in tree.getroot().iter('Model'):
44     if node.attrib.get('modelType') == 'ActivityPartition':
45         actorId = node.attrib.get('id')
46         actor = node.attrib.get('name')
47         insertActor(actorId, actor)
48
49         for item in actorTable[actorId].keys():
50             if item == 'actionIDs':
51                 for nextnode in node.iter('ModelRefsProperty'):
52                     for k in nextnode.iter('ModelRef'):
53                         actorTable[actorId]['actionIDs']\
54                                 .append(k.attrib.get('id'))
55
56 #populate actionTable
57 for node in tree.getroot().iter('Model'):
58             expectedTypes = ['ActivityAction','AcceptEventAction']
59             if node.attrib.get('modelType') in expectedTypes:
60                 actionId = node.attrib.get('id')
61                 name = node.attrib.get('name')
62                 insertAction(actionId,name)
63
64 #populate eventTable
65 for node in tree.getroot().iter('Model'):
66     if node.attrib.get('modelType') == 'ActivityAction':
```

```
67          ID = node.attrib.get('id')
68          name = node.attrib.get('name')
69          insertEvent(ID, name)
70          for nextnode in node.iter('FromSimpleRelationships'):
71              for c in nextnode.iter('RelationshipRef'):
72                  eventTable[ID]['To'].append(c.attrib.get('to'))
73      elif node.attrib.get('modelType') == 'AcceptEventAction':
74          ID = node.attrib.get('id')
75          name = node.attrib.get('name')
76          insertEvent(ID, name)
77          for nextnode in node.iter('ToSimpleRelationships'):
78              for d in nextnode.iter('RelationshipRef'):
79                  eventTable[ID]['From'].append(d.attrib.get('from'))
80
81  #build required lists now
82  exogList = []
83  instList = []
84  fluentList = []
85
86  for x in actionTable.keys():
87      for y in actorTable.keys():
88          if x in actorTable[y]['actionIDs']:
89              for z in eventTable.keys():
90                  if x == z:
91                      if eventTable[z]['To'] != eventTable[z]['From']:
92                          for k in eventTable[z]['To']:
93                              for l in actorTable.keys():
94                                  if k in actorTable[l]['actionIDs']:
95                                      if actorTable[y]['actorName'] !=
                                            actorTable[l]['actorName']:
96                                          exogList.append('ex'+actionTable[x]
                                                ['actionType']+\
97                                      '('+actorTable[y]['actorName']+','
                                            +actorTable[l]['actorName']+')')
98                                          instList.append(actionTable[x]
                                                ['actionType']+\
99                                      '('+actorTable[y]['actorName']+','
                                            +actorTable[l]['actorName']+')')
100                                         fluentList.append('f'+actionTable[x]
                                                ['actionType']+\
```

```
101                             '('+actorTable[y]['actorName']+','
                                    +actorTable[l]['actorName']+')')
102                             else:
103                                 exogList.append('ex'+actionTable[x]
                                        ['actionType']+\
104                                 '('+actorTable[y]['actorName']+')')
105                                 instList.append(actionTable[x]
                                        ['actionType']+\
106                                 '('+actorTable[y]['actorName']+')')
107                                 fluentList.append('f'+actionTable[x]
                                        ['actionType']+\
108                                 '('+actorTable[y]['actorName']+')')
109                 else:
110                     exogList.append('ex'+actionTable[x]['actionType']+\
111                                 '('+actorTable[y]['actorName']+')')
112                     instList.append(actionTable[x]['actionType']+\
113                                 '('+actorTable[y]['actorName']+')')
114                     fluentList.append('f'+actionTable[x]['actionType']+\
115                                 '('+actorTable[y]['actorName']+')')
116
117                 for k in eventTable[z]['From']:
118                     for l in actorTable.keys():
119                         if k in actorTable[l]['actionIDs']:
120                             if actorTable[y]['actorName'] !=
                                    actorTable[l]['actorName']:
121                                 exogList.append('ex'+actionTable[x]
                                        ['actionType']+\
122                                 '('+actorTable[y]['actorName']+','
                                        +actorTable[l]['actorName']+')')
123                                 instList.append(actionTable[x]
                                        ['actionType']+\
124                                 '('+actorTable[y]['actorName']+','
                                        +actorTable[l]['actorName']+')')
125                                 fluentList.append('f'+actionTable[x]
                                        ['actionType']+\
126                                 '('+actorTable[y]['actorName']+','
                                        +actorTable[l]['actorName']+')')
127                             else:
128                                 exogList.append('ex'+actionTable[x]
                                        ['actionType']+\
129                                 '('+actorTable[y]['actorName']+')')
```

```
130                                 instList.append(actionTable[x]
                                        ['actionType']+\
131                                     '('+actorTable[y]['actorName']+')')
132                                 fluentList.append('f'+actionTable[x]
                                        ['actionType']+\
133                                     '('+actorTable[y]['actorName']+')')
134
135 #Writing InstAL specifications
136
137 #get institution name
138 for firstnode in tree.getroot().iter('Project'):
139     title = firstnode.attrib.get('name')
140     print 'institution '+title+';'
141 print
142
143 #getting types
144 print '% Types\n'
145 for x in actorTable.keys():
146     print 'type '+actorTable[x]['actorName']+';'
147 print
148 # Writing exogenous events
149 print '% Exogenous events'
150 for event in exogList:
151     print 'exogenous event '+event+';'
152 print
153 #Institutional events
154 print '% Institutional events'
155 for event in instList:
156     print 'inst event '+event+';'
157 print
158 #Creation event
159 print '% Creation event'
160 print 'create event crt_'+title+';'
161 print
162 #Fluents declaration
163 print '% Fluents declaration'
164 for fluent in fluentList:
165     print 'fluent '+fluent+';'
166 print
167 #Conventional generation
168 print '% Conventional generation'
```

```
169 for exEvent,insEvent in zip(exogList,instList):
170     print exEvent + ' generates '+insEvent+';'
171 print
172 #Consequence relations
173 print '% Consequence relations'
174 for insEvent,fluent in zip(instList,fluentList):
175     print insEvent+' initiates '+fluent+';'
176 print
177 #Initialization
178 print '%initial states'
179 print 'initially'
180 for exEvent in exogList:
181     print 'perm('+exEvent+'),'
182 for insEvent in instList:
183     print 'perm('+insEvent+'), pow('+insEvent+'),'
```

Appendix B

pyinstql Query Translator Code

Listing B.1: Query Translation Code.

```
1  #!/usr/bin/python
2
3  #-------------------------------------------------------------------------
4  # REVISION HISTORY
5  # add new entries here at the top tagged by date and initials
6  # 20130401 GDB: added a definition of disjunction that allows for
       comma
7  # 20130325 GDB: modified the 'show' option to include 'occurred'
       and 'holdsat'
8  # 20130225 GDB: added function to flatten the nested list into a
       single flat list
9  # 20130222 GDB: removed the conditionCounter function
10 # 20130220 GDB: added functions for 'observe' and 'show'
11 # 20130220 GDB: added code to accept input file at command line
       using the -i option
12 #-------------------------------------------------------------------------
13
14 from __future__ import print_function
15 import re
16 import sys
17 import ply.yacc as yacc
18 import argparse # GDB added this
19
20 sys.path.insert(0,"../..")
21
22 if sys.version_info[0] >= 3:
```

```
23      raw_input = input
24
25  # class instalparserclass():
26
27  instal_output = sys.stdout
28
29  def instql_print(p): print(p,file=instql_output)
30
31  def instql_error(p): print(p,file=sys.stderr)
32
33  def instql_warn(p): print(p,file=sys.stderr)
34
35  #show_debug = False
36  show_debug = True
37
38  def debug(*p):
39      if show_debug: print(p)
40
41  #-----------------------------------------------------------------------
42  # LEXER + PARSER for instql
43
44  reserved = {
45      'and' : 'AND',
46      'not' : 'NOT',
47      'or' : 'OR',
48      'while' : 'WHILE',
49      'after' : 'AFTER',
50      'holds' : 'HOLDS',
51      'happens' : 'HAPPENS',
52      'condition' : 'CONDITION',
53      'constraint' : 'CONSTRAINT',
54      'violates' : 'VIOLATES',
55      'show' : 'SHOW',
56      'observe' : 'OBSERVE'  # GDB added
57  }
58
59  tokens =
        ['NAME','VARIABLE','INTEGER','LPAR','RPAR','SEMI','COMMA','COLON']
        + list(reserved.values())
60
61  # Tokens
```

```
62
63  t_SEMI = r';'
64  t_COMMA = r','
65  t_COLON = r':'
66  t_LPAR = r'\('
67  t_RPAR = r'\)'
68
69  def t_NAME(t):
70      r'[a-z][a-zA-Z_0-9]*'
71      t.type = reserved.get(t.value,'NAME') # Check for reserved
            words
72      return t
73
74  def t_VARIABLE(t):
75      r'[A-Z][a-zA-Z_0-9]*'
76      t.type = reserved.get(t.value,'VARIABLE') # Check for reserved
            words
77      return t
78
79  def t_INTEGER(t):
80      # note: numbers are parsed but not converted into integers
81      r'\d+'
82      # t.value = int(t.value)
83      return t
84
85  t_ignore = " \t\r"
86
87  # Comments
88  def t_COMMENT(t):
89      r'%.*'
90      pass
91      # No return value. Token discarded
92
93  def t_newline(t):
94      r'\n+'
95      t.lexer.lineno += t.value.count("\n")
96
97  def t_error(t):
98      instql_error("Illegal character '%s'" % t.value[0])
99      t.lexer.skip(1)
100
```

```
# Build the lexer
import ply.lex as lex
lex.lex()

ast = []

def p_instqlExpr(p): #GDB added observe, show
    """ instqlExpr :
        instqlExpr : instqlExpr conditionDecl
        instqlExpr : instqlExpr constraint
        instqlExpr : instqlExpr observe
        instqlExpr : instqlExpr show
    """
    global ast
# debug("instqlExpr:")
    if len(p)<2:
        ast = []
        p[0] = []
    else:
        ast = [p[2]] + p[1]
        p[0] = ast

def p_variable_list(p):
    """ variable_list :
        variable_list : VARIABLE
        variable_list : variable_list COMMA VARIABLE
    """
# debug("variable_list:")
    if len(p)>2: p[0] = p[1] + [p[3]] # general case
    elif len(p)==2: p[0] = [p[1]] # unary case
    # nullary case

def p_identifier(p):
    # second rule not in grammar, but derives from example
    #GDB: added the more rules
    """ identifier : NAME
        identifier : VARIABLE
        identifier : INTEGER
        identifier : NAME LPAR variable_list RPAR
        identifier : NAME LPAR NAME RPAR
        identifier : NAME LPAR NAME COMMA NAME RPAR
```

```
142     """
143 # debug("identifier:")
144     if len(p)>6:
145         p[0] = [p[1],p[3],p[5]]
146     elif len(p)>4:
147         p[0] = [p[1],p[3]]
148     else:
149         p[0] = p[1]
150 
151 #GDB: added the second rule
152 def p_happens(p):
153     """ happens : HAPPENS LPAR identifier RPAR
154         happens : HAPPENS LPAR identifier COMMA identifier RPAR
155     """
156 # debug("happens:")
157     if len(p)>5:
158         p[0] = ['HAPPENS',p[3],p[5]]
159     else:
160         p[0] = ['HAPPENS',p[3]]
161 
162 def p_violates(p):
163     """ violates : VIOLATES LPAR identifier RPAR
164         violates : VIOLATES LPAR identifier COMMA identifier RPAR
165     """
166 # debug("violates:")
167     if len(p)>5:
168         p[0] = ['VIOLATES',p[3],p[5]]
169     else:
170         p[0] = ['VIOLATES',p[3]]
171 
172 def p_holds(p):
173     """ holds : HOLDS LPAR identifier RPAR
174         holds : HOLDS LPAR identifier COMMA identifier RPAR
175     """
176 # debug("happens:")
177     if len(p)>5:
178         p[0] = ['HOLDS',p[3],p[5]]
179     else:
180 # debug("holds:")
181         p[0] = ['HOLDS',p[3]]
182 
```

```
183 def p_literal(p):
184     """ literal : NOT happens
185         literal : NOT holds
186         literal : happens
187         literal : holds
188         literal : violates
189     """
190 # debug("literal:")
191     if len(p)>2:
192         p[0] = ['NOT',p[2]]
193     else:
194         p[0] = p[1]
195 
196 def p_whileExpr(p):
197     """ whileExpr : literal
198         whileExpr : literal WHILE whileExpr
199     """
200 # debug("whileExpr:")
201     if len(p)>3:
202         p[0] = ['WHILE',p[1],p[3]]
203     else:
204         p[0] = p[1]
205 
206 def p_after(p):
207     """ after : AFTER
208         after : AFTER LPAR INTEGER RPAR
209     """
210 
211     if len(p)>2:
212         debug("after:",p[3])
213         p[0] = ['AFTER',p[3]]
214     else:
215 # debug("after:")
216         p[0] = 'AFTER'
217 
218 def p_afterExpr(p):
219     """ afterExpr : whileExpr
220         afterExpr : whileExpr after afterExpr
221     """
222 # debug("afterExpr:")
223     if len(p)>3:
```

```
224         if p[2]=='AFTER':
225             p[0] = [p[2],p[1],p[3]]
226         else:
227             p[0] = [p[2][0],p[2][1],p[1],p[3]]
228     else:
229         p[0] = p[1]
230 
231 def p_conditionLiteral(p):
232     """ conditionLiteral : NOT identifier
233         conditionLiteral : identifier
234         conditionLiteral : identifier LPAR identifier COMMA
                identifier RPAR
235     """
236 # debug("conditionLiteral:")
237     if len(p)>2:
238         p[0] = ['NOT',p[2]]
239     else:
240         p[0] = p[1]
241 
242 def p_term(p):
243     """ term : afterExpr
244         term : conditionLiteral
245     """
246 # debug("term:")
247     p[0] = p[1]
248 
249 def p_conjunction(p):
250     """ conjunction : term
251         conjunction : conjunction AND term
252     """
253 # debug("conjunction:")
254     if len(p)>3:
255         p[0] = ['AND',p[1],p[3]]
256     else:
257         p[0] = p[1]
258 
259 def p_disjunction(p): #GDB: added a rule that accepts comma
260     """ disjunction : conjunction
261         disjunction : disjunction COMMA conjunction
262         disjunction : disjunction OR conjunction
263     """
```

```
264 # debug("disjunction:")
265     if len(p)>3:
266         p[0] = ['OR',p[1],p[3]]
267     else:
268         p[0] = p[1]
269 
270 def p_conditionDecl(p):
271     """ conditionDecl : CONDITION term COLON disjunction SEMI
272     """
273 # debug("conditionDecl:")
274     p[0] = ['CONDITION',p[2],p[4]]
275 
276 def p_constraint(p):
277     """ constraint : CONSTRAINT disjunction SEMI
278     """
279 # debug("constraint:")
280     p[0] = ['CONSTRAINT',p[2]]
281 
282 # GDB: functions for observe and show
283 def p_observe(p):
284     """ observe : OBSERVE disjunction SEMI
285     """
286 # debug("observe:")
287     p[0] = ['OBSERVE',p[2]]
288 
289 def p_show(p):
290     """ show : SHOW disjunction SEMI
291     """
292 # debug("show:")
293     p[0] = ['SHOW',p[2]]
294 
295 def p_error(p):
296     if p:
297         debug("Syntax error at '%s'" % p.value)
298     else:
299         debug("Syntax error at EOF")
300 
301 #def term2string(p):
302 # # print "term2string: p = ",p
303 # args = p[1]
304 # r=''
```

```
305 # if len(args)==0:
306 # r=p[0]
307 # elif len(args)==1:
308 # r=p[0]+'('+args[0]+')'
309 # else:
310 # r='('+args[0]
311 # for x in args[1:]: r=r+','+x
312 # r=p[0]+r+')'
313 # return r
314
315 #-----------------------------------------------------------------------
316
317 def instql_parse(d):
318     yacc.yacc()
319     yacc.parse(d)
320
321 #GDB 20130220: code for file input at command line
322 parser = argparse.ArgumentParser()
323 parser.add_argument("-i","--input-file")
324 args = parser.parse_args()
325
326 inp = open(args.input_file,'r')
327
328 document = " "
329
330 if args.input_file: document = inp.read(-1)
331 else: document = sys.stdin.read(-1)
332 #debug("input: ",document)
333 instql_parse(document)
334 ast.reverse()
335 #debug("output: ",ast)
336 #debug("--------------------------")
337
338 def id2string(p): #GDB: Modified to treat arguments as strings
339     #debug("is2string: p = ",p)
340     if isinstance(p,str): return p
341     args = p[1:]
342     #debug('args = ',args)
343     r=''
344     if isinstance(args,str):
345         r=p[0]+'('+args+')'
```

```
346     else:
347         if len(args)>1:
348             r=p[0]+'('+args[0]+','+args[1]+')'
349         else: r=p[0]+'('+args[0]+')'
350     return r
351 
352 instantCounter = 0
353 
354 def newInstant():
355     global instantCounter
356     (instantCounter)+=1
357     return `instantCounter`
358 
359 def thisInstant():
360     global instantCounter
361 # debug('thisInstant =',instantCounter)
362     return `instantCounter`
363 
364 def flatten(L): #GDB: Added this function to flatten out nested
        lists
365     if not L:
366         return L
367     elif isinstance(L,str):
368         return L
369     elif type(L[0]) == type([]):
370         return flatten(L[0]) + flatten(L[1:])
371     else:
372         return [L[0]] + flatten(L[1:])
373 
374 def param(a): #GDB: Added this function to handle parameters
375     b = a[0]
376     if isinstance(b,str):
377         return a[0]
378     else:
379         if len(b)>2:
380             return b[0]+'('+b[1]+','+b[2]+')'
381         else:
382             return b[0]+'('+b[1]+')'
383 
384 def instql_print(t):
```

```
385     instantCounter = int(instantCounter) # GDB: This is somehow
            messy
386     newInstant()
387     if t==[]: return ""
388     k=t[0]
389     a=t[1:]
390     if k=='WHILE': #GDB: modified for handling instants
391         global instantCounter
392         z = instantCounter
393         r = instql_print(a[0])
394         instantCounter = z
395         for x in a[1:]: r += ',' + instql_print(x)
396         return r
397     if k=='AFTER': #GDB: modified and extended to handle instants
            properly
398         d = ''
399         if (len(a)==3):
400             d = a[0]
401             a = a[1:]
402             r = instql_print(a[0])
403             for x in a[1:]:
404                 z = thisInstant()
405                 r += ',' + instql_print(x) + (
406                     ',after({i1},{i2},{d})'
407                     .format(i1='I'+z,i2='I'+thisInstant(),d=d))
408         else:
409             r = instql_print(a[0])
410             for x in a[1:]:
411                 z = thisInstant()
412                 w = instql_print(x)
413                 r += ',' + w + (',after({i1},{i2})'
414                     .format(i1='I'+z,i2='I'+str(instantCounter)))
415                 instantCounter = z
416         return r
417
418     if k=='HAPPENS':
419         c = param(a)
420         if len(a)>1:
421             return('occurred({ev},{inst}),event({ev}))'
422                .format(ev=c,inst=a[1]))
423         else:
```

```
424             return('occurred({ev},{inst}),event({ev}),instant({inst})'
425                .format(ev=c,inst='I'+thisInstant()))
426
427     if k=='VIOLATES': #GDB: Added this option to handle violations
428         c = param(a)
429         if len(a)>1:
430             return('occurred(viol({ev}),{inst}),event({ev})'
431                .format(ev=c,inst=a[1]))
432         else:
433             return('occurred(viol({ev}),{inst}),event({ev}),instant({inst})'
434                .format(ev=c,inst='I'+thisInstant()))
435
436     if k=='HOLDS':
437         c = param(a)
438         if len(a)>1:
439             if a[1] == 'F':
440                 return('holdsat({f1},{inst}),ifluent({f1}),final({inst}))'
441                .format(f1=c,inst=a[1]))
442             else:
443                 return('holdsat({f1},{inst}),ifluent({f1}))'
444                        .format(f1=c,inst=a[1]))
445         else:
446             return('holdsat({fl},{inst}),ifluent({fl}),instant({inst})'
447                .format(fl=c,inst='I'+thisInstant()))
448     if k=='NOT':
449         return('not {pred}'
450                .format(pred=instql_print(a[0])))
451     if k=='OR':
452         return [instql_print(x) for x in a]
453     if k=='AND':
454         r = instql_print(a[0])
455         for x in a[1:]: r += ',' + instql_print(x)
456         return r
457     if k=='CONDITION':
458         r = ''
459         s = instql_print(a[1])
460         f = flatten(s)
461         # check for string or list
462         if isinstance(f,str):
463             r = ('{name} :- {body}.\n'
464                  .format(name=id2string(a[0]),body=f))
```

```
465         else:
466             for x in f:
467                 r += ('{name} :- {body}.\n'
468                       .format(name=id2string(a[0]),body=x))
469         return r
470     if k=='CONSTRAINT': #GDB: modified and extended to handle list
        of constraints and parameters
471 # debug('constraint',a)
472         r = ''
473         s = instql_print(a[0])
474         f = flatten(s)
475         if isinstance(f,str): # check for string or list
476             if (f[0:3] == 'not'):
477                 r = (':- {body}.\n'
478                  .format(body=f[3:]))
479             else:
480                 r = (':- not {body}.\n'
481                  .format(body=f))
482         else:
483             for x in f:
484                 if (x[0:3] == 'not'):
485                     r += (':- {body}.\n'
486                      .format(body=x[3:]))
487                 else:
488                     r += (':- not {body}.\n'
489                        .format(body=x))
490         return r
491
492     # GDB: prints the observed events
493     if k=='OBSERVE':
494         #c = param(a)
495         #debug('c:',c)
496         r = ''
497         s = instql_print(a[0])
498         #debug('s:',s)
499         f = flatten(s)
500         #debug('f:',f)
501         if isinstance(f,str):
502             instantCounter = 0
503             r = ('observed({ev}, {inst}).\n'
504                  .format(ev=f, inst=thisInstant()))
```

```
505
506         else:
507             instantCounter = 0
508             for x in f:
509                 r += ('observed({ev}, {inst}).\n'
510                             .format(ev=x, inst=thisInstant()))
511
512                 instantCounter+=1
513         return r
514     # GDB: prints the show conditions
515     if k=='SHOW':
516         r = ''
517         r = '#hide.\n'
518         s = instql_print(a[0])
519         f = flatten(s)
520         if isinstance(f,str):
521             if f == 'events':
522                 r += ('#show {ev}.\n'
523                       .format(ev='occurred(E,I)'))
524             elif f == 'states':
525                 r += ('state(F,I):- instant(I),fluent(F).\n'
526                       '#show {ev}.\n'
527                       .format(ev='state(F,I)'))
528             elif f == 'violations':
529                 r += ('#show {ev}.\n'
530                       .format(ev='occurred(viol(E),I)'))
531             else:
532                 r += ('#show {ev}.\n'
533                      .format(ev=f))
534         else:
535             for x in f:
536                 if x == 'events':
537                     r += ('#show {ev}.\n'
538                           .format(ev='occurred(E,I)'))
539                 elif x == 'states':
540                     r += ('state(F,I):- instant(I),fluent(F).\n'
541                       '#show {ev}.\n'
542                       .format(ev='state(F,I)'))
543                 elif x == 'violations':
544                     r += ('#show {ev}.\n'
545                       .format(ev='occurred(viol(E),I)'))
```

```
546                 else:
547                     r += ('#show {ev}.\n'
548                         .format(ev=x))
549         return r
550
551
552     return id2string(t)
553
554 for x in ast: print(instql_print(x))
```

Appendix C

pyviz Answer Set Visualiser Code

Listing C.1: Answer Set Visualiser Code.

```
1  #!/usr/bin/python
2
3  #--------------------------------------------------------------------------
4  # REVISION HISTORY
5  # add new entries here at the top tagged by date and initials
6  # GDB 201305??: added input file argument and consequent changes
7  # GDB/JAP?????: changed main loop to iterate over occurred not
       holdsat
8  # JAP 201305??: first version (approx)
9
10 from __future__ import print_function
11 import re
12 import sys
13 import ply.lex as lex
14 from collections import defaultdict
15 import string
16 from itertools import izip
17 from itertools import count
18 import argparse
19
20 class myLexer():
21
22     # Build the lexer
23     # def build(self,**kwargs):
24     # self.lexer = lex.lex(object=self, **kwargs)
25
```

```
26      def __init__(self):
27          self.lexer = lex.lex(module=self)
28
29      reserved = { }
30
31      tokens = ['NAME','NUMBER','LPAR','RPAR','COMMA']
32
33      # Tokens
34
35      t_COMMA = r','
36      t_LPAR = r'\('
37      t_RPAR = r'\)'
38
39      def t_NAME(self,t):
40          r'[a-z][a-zA-Z_0-9]*'
41          return t
42
43      def t_NUMBER(self,t):
44          r'\d+'
45          # t.value = int(t.value)
46          return t
47
48      t_ignore = " \t\r"
49
50      # Comments
51      def t_COMMENT(self,t):
52          r'%.*'
53          pass
54      # No return value. Token discarded
55
56      def t_newline(self,t):
57          r'\n+'
58          t.lexer.lineno += t.value.count("\n")
59
60      def t_error(self,t):
61          print("Illegal character '%s'" % t.value[0])
62          t.lexer.skip(1)
63
64  def pyvizError(s):
65      print(s,file=sys.stderr)
66
```

```
67  observed = defaultdict(list)
68  holdsat = defaultdict(list)
69  initiated = defaultdict(list)
70  terminated = defaultdict(list)
71  occurred = defaultdict(list)
72  state = defaultdict(list)
73  compromised = defaultdict(list)
74
75  def processHoldsat(l):
76      # lots of tacky dead-reckoning :(
77      time = int(l[-2]) # -2 is time instant
78      holdsat[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever holdsat
79
80  def processState(l):
81      time = int(l[-2]) # -2 is time instant
82      state[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever state
83
84  def processCompromised(l):
85      time = int(l[-2]) # -2 is time instant
86      state[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever state
87
88  def processObserved(l):
89      time = int(l[-2]) # -2 is time instant
90      observed[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever observed
91
92  def processInitiated(l):
93      time = int(l[-2]) # -2 is time instant
94      initiated[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever initiated
95
96  def processTerminated(l):
97      time = int(l[-2]) # -2 is time instant
98      terminated[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',',
            ')) # should be whatever terminated
99
100 def processOccurred(l):
101     time = int(l[-2]) # -2 is time instant
```

```
102     occurred[time].append(string.join(l[2:-3],'').replace('_','\_').replace(',',',
            ')) # should be whatever terminated
103
104 # command line arguments
105
106 def arb(s): return s
107
108 parser = argparse.ArgumentParser()
109 parser.add_argument("-a", "--answer-set", type=arb,
110                     help="specify answer set (default 1)")
111 # GDB 20130430
112 parser.add_argument("-i", "--answerset-file", type=arb,
113                     help="specify answer set file")
114
115 args=parser.parse_args()
116
117 answer_set="1"
118 if args.answer_set:
119     answer_set = args.answer_set
120 # GDB 20130430
121 if args.answerset_file:
122     f = open(args.answerset_file,'r')
123
124 document = ""
125
126 if args.answerset_file:
127     document = f #document + f.read(-1)
128 else:
129     document = sys.stdin #document + sys.stdin.read(-1)
130 #debug('line:',document)
131
132 mylex=myLexer()
133 for line in document:#sys.stdin:
134     if re.match("Answer: {n}".format(n=answer_set),line): break
135 for line in document:#sys.stdin:
136     # split line into terms so that process can group output about
            each
137     # mechanism found
138     found = True
139     for term in re.split(' ',line):
140         mylex.lexer.input(term)
```

```
141         l = [tok.value for tok in mylex.lexer]
142         if l==[]: continue # skip blanks
143         if l[0]=='holdsat':
144             processHoldsat(l)
145         elif l[0]=='state':
146             processState(l)
147         elif l[0]=='compromised':
148             processState(l)
149         elif l[0]=='observed':
150             processObserved(l)
151         elif l[0]=='initiated':
152             processInitiated(l)
153         elif l[0]=='terminated':
154             processTerminated(l)
155         elif l[0]=='occurred':
156             processOccurred(l)
157         else:
158             print("% skipping \"{term}\""
159                   .format(term=string.join(l,'')))
160     break # stop after specified answer set
161 if not found:
162     print("Answer set {n} not found".format(n=answer_set))
163     exit(-1)
164 # output the latex
165 print("\\resizebox{\\textwidth}{!}{\n"
166       "\\begin{tikzpicture}\n"
167       "[\nstart chain=trace going right,")
168 # establish how many states there are
169 event_count=max(len(occurred),len(observed))
170 inst_count=max(len(holdsat),len(initiated),len(terminated),len(state))
171 nstates=event_count+1 if event_count>0 else inst_count
172 # set up state chains
173 for t in range(0,nstates):
174     print("start chain=state{i} going down,".format(i=t))
175 print("node distance=1cm and 5.2cm\n]")
176 for t in range(0,nstates):
177     print("{{{{ [continue chain=trace]\n"
178           "\\node[circle,draw,on chain=trace] (i{i})
                 {{$S_{{{i}}}$}};"
179           .format(i=t))
180     if (t>0):
```

```
181         # connectors between states labelled with observed and
                occurred events
182         print("\draw[-latex](i{i}) -- \n"
183               .format(i=t-1)
184               +
                      "node[above]{\\begin{tabular}{>{\centering}m{5cm}}\n"
185               + string.join(observed[t-1],"\\\\\n")
186               + "\\\\\n\\em "
187               + string.join(occurred[t-1],"\\\\\n\\em ")
188               + "\n\\end{tabular}")
189         print("}}\n(i{i});"
190               .format(i=t))
191     print("}}")
192     if
            max(len(holdsat[t]),len(initiated[t]),len(terminated[t]),len(state[t]),
            len(compromised[t]))>0:
193         print("{{ [continue chain=state{i} going below]\n"
194               "\\node [on chain=state{i},below=of
                      i{i},rectangle,draw,rounded corners,inner frame
                      sep=0pt] (s{i}) {{\n"
195               "% instant {i}"
196               .format(i=t))
197         # table of fluents for each state
198         # was
                "\\begin{tabular}[t]{|r@{\hspace{3pt}}p{5cm}|}\\hline\n"
199         # was
                "\\begin{tabular}[t]{|@{\hspace{1pt}}p{0.3cm}@{}p{5cm}|}\\hline\n"
200         print("\\begin{tabular}[t]{@{}p{5cm}@{}}\n"
201               + string.join(["\\sout{"+x+"}\\\\\n" for x in
                      terminated[t]],'')
202               + ("\\hline " if len(terminated[t])!=0 else "")
203               + string.join([x+"\\\\\n" if (t>0) and
204                              ((x in holdsat[t-1]) or (x in
                                   initiated[t-1])) else
205                              "\\textbf{\\textcolor{blue}{"+x+"}}\\\\\n"
                                   # bold new fluents
206                              for x in holdsat[t] if x not in
                                   terminated[t]],'')
207               + string.join([x+"\\\\\n" if (t>0) and
208                              ((x in state[t-1]) or (x in
                                   initiated[t-1])) else
```

```
209                             "\\textbf{\\textcolor{blue}{"+x+"}}\\\\\n"
                                    # bold new fluents
210                             for x in state[t] if x not in
                                    terminated[t]],'')
211             + string.join([x+"\\\\\n" if (t>0) and
212                             ((x in compromised[t-1]) or (x in
                                    initiated[t-1])) else
213                             "\\textbf{\\textcolor{blue}{"+x+"}}\\\\\n"
                                    # bold new fluents
214                             for x in compromised[t] if x not in
                                    terminated[t]],'')
215             + ("\\hline " if len(initiated[t])!=0 else "")
216             +
                    string.join(["\\textbf{\\textcolor{blue}{"+x+"}}\\\\\n"
                    for x in initiated[t]],'')
217             + "\\end{tabular}\n};")
218         print("} % end node and chain")
219         print("\draw (i{i}) -- (s{i});\n"
220               .format(i=t))
221     print("% \pause % uncomment here to animate\n")
222 # provenance of trace
223 print("\\draw(i0)+(-3,0)node[rotate=90]{{Answer set {i}, {f}}};"
224       .format(i=answer_set,
225               f=args.answerset_file if args.answerset_file else
                    'stdin'))
226 print("\\end{tikzpicture}\n}\n")
227
228 if args.answerset_file: f.close()
```